Minimal realization of sequential machines : the method of maximal adjacencies

Citation for published version (APA):

Document status and date:
Published: 01/01/1988

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Minimal Realization of Sequential Machines: The Method of Maximal Adjacencies

by

L. Józwiak

EUT Report 88-E-209
November 1988
MINIMAL REALIZATION OF SEQUENTIAL MACHINES:
The method of maximal adjacencies

by

L. Jóźwiak

EUT Report 88-E-209

Eindhoven
November 1988
Minimal realization of sequential machines: the method of maximal adjacencies / by L. Jóźwiak. - Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig. - (EUT report, ISSN 0167-9708; 88-E-209)

Met lit. opg., reg.
SISO 664 UDC 681.325.65:519.6 NUGI 832
Trefw.: automatentheorie.
MINIMAL REALIZATION of SEQUENTIAL MACHINES

THE METHOD OF MAXIMAL ADJACENCIES

Lech Jóźwiak

Group Digital Systems, Faculty of Electrical Engineering, Eindhoven University of Technology (The Netherlands)

Abstract—Reducing the amount of hardware needed for implementing a sequential machine is very important.

The problem of, in the strict sense, minimal realization of sequential machines is difficult and it is entirely unsolved (except complete enumeration).

Typically, this problem can be replaced with a number of subproblems such as: state minimization, state assignment, choice of flip-flops and minimization of Boolean functions representing the next-state and output functions of a sequential machine.

In this work, the greatest attention has been paid to state assignment and state minimization; however, the method, as a whole, covers all the subjects listed.

Our aim was to find a practical method of state assignment for medium and large sequential machines.

Two traditionally independent steps: state minimization and state assignment are replaced here with a single process of concurrent state minimization and assignment. Here, minimization (or partial minimization) of internal states is obtained as a byproduct of the state assignment and it results from assigning the same code to two or more internal states.

The problem of, in the strict sense, optimal state assignment is unsolved, but some approximate approaches have been proposed. The best known of them are: the partition theory [7][8][12][13][16][22][24][25], the column based approach [4], the graph embedding approach [1][2][15] and related to it multi-valued, multi-output, non-univocal function minimization methods [3][20][21].

Two first approaches suffer from many shortcomings.

The method presented here is related to the third group of methods. Some of the observations, on which our method is based, are like those used by Armstrong [1][2]. However, many important differences exist between the method presented here and Armstrong's method.

The method of maximal adjacencies uses much more information about the factors, that can have influence on the quality of the resultant assignments than the method of Armstrong [1][2] and, also, than all the other related methods [3][15][20][21]. Therefore, in many cases, it can produce better assignments than the methods of the third group. For the same reason, it can give better assignments than the column based approach.

The method presented here uses adjacency conditions, that are ordered according to the number of adjacencies reached when a given condition is satisfied by the assignment. The number of adjacencies reflects the condition's quality. Since the conditions are considered and combined, starting with the best, the first assignments constructed will be always nearly optimal and, almost always, the best of the nearly optimal solutions will be one of the first to be obtained.
In the methods using minimization of multi-valued, multi-output non-univocal functions for creating the conditions used further in order to construct the near optimal assignments, such a measure of quality of conditions and appropriate ordering relation on the set of conditions did not exist. Thus, the method of maximal adjacencies seems to be most effective, i.e. it produces good results more quickly. The capacity of this method is very important, especially, for large machines with "difficult" algebraic structures, for which the construction of assignments is time and memory consuming.

The method does not assume minimum numbers of states and memory elements.

Furthermore, some of the best types of flip-flops can be adopted in order to realize each excitation function and the complexity of the realization of the output function is taken into account.

The method of maximal adjacencies contains none of the shortcomings of the first two state assignment approaches.

Index Terms- Automata theory, logic minimization, logic system design, sequential machines.

Acknowledgement- The author is indebted to Prof.ir. A. Heetman and Prof.ir. M.P.J. Stevens for making it possible to perform this work, to Dr. P.R. Attwood for making corrections to the English text and to mr. C. van de Watering for typing the text.
1. Introduction 1

2. The silicon area used by a sequential machine and the state assignment problem 2

3. Partitions and the state assignment 13

4. The method of maximal adjacencies 20
   4.1 The adjacency conditions for input-state dependencies 22
   4.2 The adjacency conditions for present-state--next-state dependencies 26
   4.3 The adjacency conditions for state-output dependencies 29
   4.4 Combined adjacency conditions 31
   4.5 Ordering of adjacency conditions 34
   4.6 Constructing the final families of partitions based on the ordered list of adjacency conditions 38
   4.7 The method of maximal adjacencies for unminimal machines 54
   4.8 Conditions for common terms 77
   4.9 Dynamic ordering of adjacency conditions 84

5. Conclusions 85

LITERATURE 89
1. Introduction

Most of the architectures of today's digital systems implement Glushkov's model of information processing.

In these architectures, it is possible to distinguish two basic parts:
- an operative unit, implementing tools for performing operations with the data,
- a control unit, implementing control algorithms of a given information processing system.

The control units of large digital systems can engage up to 80% of the entire hardware implementing the systems and, therefore, it is very important to reduce the amount of hardware used by the control unit. The reduction of the area occupied by the control unit gives additionally shorter signal paths and, in consequence, improves performance.

The control unit can be constructed as a sequential machine (finite state machine, finite automaton) or a number of cooperating sequential machines.

Reducing the amount of hardware needed for implementing a sequential machine is a complicated process which can be brought into effect in a number of steps implementing some optimization algorithms.

These steps include: state reduction, state assignment, choice of flip-flops, minimization of the Boolean functions representing the next-state and output functions of a sequential machine.

In this work, the attention is paid to the problem of the optimal state assignment, but the method presented, as a whole, covers all the subjects listed.
2. The silicon area used by a sequential machine and the state assignment problem.

**Definition 2.1.** A sequential machine $M$ is an algebraic system defined as follows:

$$M = (I, S, O, \delta, \lambda),$$

where:

$I$ - finite nonempty set of inputs,
$S$ - finite nonempty set of internal states,
$O$ - finite set of outputs,
$\delta$ - next state function, $\delta : S \times I \rightarrow S$,
$\lambda$ - output function, $\lambda : S \times I \rightarrow O$ (a Mealy machine)

or

$\lambda : S \rightarrow O$ (a Moore machine)

If the output set $O$ and the output function $\lambda$ are not defined, the sequential machine $M = (I, S, \delta)$ is called a **state machine**.

A sequential machine can be implemented according to the different implementation strategies using different building blocks for synthesizing the logic. The most common approaches are the following: random logic implementation (logic is constructed from a set of gates, flip-flops etc.), array logic implementation (logic is achieved by the suitable programming of a single array logic block or a set of array logic blocks such as PLS, PLA, PAL, PGA) microprogramming approach (control memory is implemented with ROM and complex decoding and sequencing operations with PLA, registers and/or counters or with another specialized structure).

In the case of implementation strategies using the random logic or the array logic, a sequential machine is completely hardware-implemented. In this case, the hardware implementation of a sequential machine requires two components:
- combinational logic;
- state memory (Fig. 2.1).
Fig. 2.1. A general model of hardware implementation for a sequential machine.

Combinational logic realizes the next-state function $\delta$ and output function $\lambda$. It can be implemented with random logic, but array logic is also a very good structure for implementing this component.

State memory is implemented through binary memory elements—in most cases through clocked delay flip-flops (D flip-flops).

Now, we will consider the random logic (RL) two-level AND-OR realization of the combinational component of a sequential machine and the programmable logic array (PLA) AND-OR realization.

The silicon area used to implement the combinational part with random logic can be approximated by:

**RL silicon area** =

$$\text{circuit area} + \text{peripheral area} + \text{overall overhead}$$

$$\text{circuit area} = u_e \cdot \sum_{i=1}^{n_e} n_{e_i} + u_e \cdot n_e$$
peripheral area = $u_{i} \cdot n_{i} + u_{o} \cdot n_{o}$
overall area = overhead due to ground lines, power lines etc.

where:

$u_{e}$ - unit silicon area for the logic element (AND or OR) per single input of the logic element,

$n_{i_{1}}$ - number of inputs to logic element 1 of the circuit,

$n_{e}$ - number of elements in the circuit,

$u_{c}$ - unit silicon area for the connections between two elements in the circuit,

$n_{c}$ - number of connections between the elements in the circuit,

$u_{i}$ - unit silicon area per input,

$n_{i}$ - number of circuit inputs,

$u_{o}$ - unit silicon area per output,

$n_{o}$ - number of circuit outputs.

In two-level AND-OR realization:

$$n_{e} = n_{OR} + n_{AND},$$

where:

$n_{OR}$ - number of OR gates,

$n_{AND}$ - number of AND gates.

Further:

$$n_{OR} = \text{number of logic functions realized} = n_{o},$$

$$n_{AND} = n_{F},$$
where:

\( n_{PT} \) - number of product terms in all the functions realized.

So, we have: \( n_e = n_{PT} + n_o \),

\[
\sum_{i=1}^{n_e} n_{ie} = \sum_{j=1}^{n_{AND}} n_{AND_j} + \sum_{k=1}^{n_{OR}} n_{OR_k},
\]

\[
n_e = \sum_{k=1}^{n_{OR}} n_{OR_k},
\]

where:

\( n_{AND_j} \) - number of inputs to an AND element \( j \),

\( n_{OR_k} \) - number of inputs to an OR element \( k \).

Further:

\[
n_{AND_j} = n_j - O_{PT_j},
\]

\[
n_{OR_k} = n_{APT_k},
\]

where:

\( O_{PT_j} \) - order of the product term \( j \),

\( n_{APT_k} \) - number of active product terms for a Boolean function \( k \) in the Boolean realization of the next-state and output functions \( \delta \) and \( \lambda \).

So we have:

\[
\sum_{i=1}^{n_e} n_{ie} = \sum_{j=1}^{n_{PT}} (n_j - O_{PT_j}) + \sum_{k=1}^{n_{APT}} n_{APT_k},
\]

\[
n_e = \sum_{k=1}^{n_{APT}} n_{APT_k}.
\]
where:

\( N_I \) - number of input variables of the sequential machine,

\( N_S \) - number of state variables of the sequential machine,

\( N_O \) - number of output variables of the sequential machine.

So, the silicon area used for implementing the combinational part of the sequential machine with random logic can be approximated by:

\[
RLSA = u_e \cdot \left( \sum_{j=1}^{n_{PT}} (N_I + N_S - O_{PT_j}) + \sum_{k=1}^{N_O + N_S} n_{APT_k} \right) + \sum_{k=1}^{N_O + N_S} u_c \cdot n_{APT_k} + u_i (N_I + N_S) + u_o (N_O + N_S) + \text{overall overhead.}
\]

The silicon area used for implementing the combinational logic of the sequential machine with PLA can be approximated by:

\[
PLA_{SL} = \text{matrix area} + \text{peripheral area} + \text{overall overhead},
\]

\[
\text{matrix area} = (u_i \cdot n_I + u_o \cdot n_o) \cdot n_p,
\]

\[
\text{peripheral area} = O_I \cdot n_I + O_o \cdot n_o + O_p \cdot n_p,
\]

where:
\[ u_i - \text{unit silicon area per matrix input}, \]
\[ n_i - \text{number of matrix inputs}, \]
\[ u_o - \text{unit silicon area per matrix output}, \]
\[ n_o - \text{number of matrix outputs}, \]
\[ O_i - \text{unit overhead per input decoder}, \]
\[ O_o - \text{unit overhead per output buffer}, \]
\[ O_p - \text{unit overhead for the pull-up and interfaces}. \]

Because:
\[ n_i = N_I + N_s, \]
\[ n_o = N_0 + N_s \]
and
\[ n_p = n_{pT}. \]

The silicon area used by PLA for implementing the combinational logic of the sequential machine can be approximated by:

\[ \text{PLA}_{sA} = (u_i(N_I + N_s) + u_o(N_0 + N_s)) \cdot n_{pT} + \]
\[ + O_i \cdot (N_I + N_s) + O_o \cdot (N_0 + N_s) + O_p \cdot n_{pT} + \]
\[ + \text{overall overhead}. \]

The silicon area used for implementing the state memory of the sequential machine can be approximated by:

\[ \text{SM}_{sA} = u_s \cdot N_s, \]

where:

\[ u_s - \text{unit silicon area used by the memory element implementing one state variable}, \]
\[ N_s - \text{number of state variables}. \]

The state assignment consists of choosing a Boolean representation for the internal states of a sequential machine, i.e. choosing a number of Boolean state variables (representing binary memory elements) and combinations of values for these variables which represent the states of a sequential machine.

Analyzing the above approximations of the silicon area for different implementations of sequential machines, we can see that the choice of codes for states effects substantially the complexity of the state memory, as well as, the complexity of the combinational logic. So, the optimal state assignment is a very
important design problem.

The complexity of the state memory is proportional to the state assignment length, i.e. the number of Boolean state variables $N_s$ in the Boolean representation of the states.

The complexity of the combinational logic depends, in a much more complicated manner, on the state assignment.

In the case of the random logic implementation, such parameters as: the number of state variables $N_s$ (state assignment length), the number of product terms in the realization of the next-state and output functions $n_{PT}$, the number of product terms active for each of these functions $n_{PT}$ and the order of product terms $OPT_j$ are affected by the state assignment. For the random logic realization, the silicon area $RL_{sc}$ used by the combinational part grows with: $N_s$, $n_{PT}$ and $n_{PT}$ and decreases with $OPT_j$. So, in order to minimize the silicon area $RL_{sc}$, we should minimize the complicated function that grows with the state assignment length, the number of product terms, the number of active product terms and the number of variables in each product term. The way that $RL_{sc}$ depends upon these parameters differs for each parameter and, additionally, interrelationships between these parameters occur.

For PLA implementation, the number of state variables (state assignment length) $N_s$ and the number of product terms used for realizing the next-state and output functions $n_{PT}$ are both affected by the state assignment. The silicon area $PLA_{sc}$ used by the combinational part grows with $N_s$ as well as with $n_{PT}$. So, in order to minimize the silicon area $PLA_{sc}$, we should minimize the function that grows with the state assignment length and the number of product terms.

In this case, the silicon area is independent of a number of active product terms for each function and a number of variables in each product term.

In general, the problem of the optimal state assignment can be stated as follows:

to find a state assignment which minimizes the total silicon area used to implement the state memory and the combinational logic of the sequential machine.

Because of the complicated influence of the state assignment
on the silicon area used to implement the sequential machine, we should try to make this problem more simple.

From approximations of the silicon area given above, it follows that the state assignment length $N_s$ influences the silicon area in very many ways. So, it is reasonable to think, that this parameter, in most cases, has the greatest influence on the silicon area.

This conclusion also follows from an analysis of many examples.

So, in order to find the optimal state assignment, we can try to minimize the assignment length first and, then, to consider the other parameters that have an influence on the complexity of the implementation of a sequential machine. We must remember, however, that in some cases we have also to take into account assignments with a length greater than the minimum.

In this way, for the PLA realization, we will obtain the following criterion:

find an assignment that minimizes the number of product terms among the assignments with a given assignment length.

An assignment with the minimal length minimizes only the complexity of the state memory and the number of inputs, outputs and columns in PLA. Minimization of the number of PLA's rows means, in fact, the minimization of the number of product terms in the Boolean realization of the next-state and output functions which can be achieved by simplifying the Boolean representations of the functions $δ$ and $λ$ in order to obtain functions with a minimal number of product terms.

For longer assignments, the complexity of the state memory and the number of inputs, outputs and columns in PLA will be greater, but the Boolean representations of the functions $δ$ and $λ$ may sometimes be constructed with a substantially smaller number of product terms and, therefore, the assignments with a nonminimal assignment length are sometimes optimal.

In the case of random logic implementations, the solution is much more complicated, because, for a given assignment length, we should minimize not only the number of product terms, but the function:
\[ R_{L_{SA}} = u_e \cdot \left( \sum_{j=1}^{n_{PT}} (N_I + N_S - O_{PT_j}) + \sum_{k=1}^{W_0 + W_S} n_{APT_k} \right) + \]
\[ + u_c \cdot \sum_{k=1}^{W_0 + W_S} n_{APT_k} + u_1 (N_I + N_S) + u_0 (N_0 + N_S) + \]
\[ + \text{overall overhead.} \]

This is equivalent to the minimization of the function:

\[ \text{RL}_{SA} = u_e \cdot \sum_{j=1}^{n_{VP}} n_{VP_j} + (u_e + u_c) \cdot \sum_{k=1}^{n_{APT}} n_{APT_k} + C ; \]

where:

- \( n_{VP_j} \) - the number of variables in the product term \( j \),
- \( N_F = N_0 + N_S \) - the number of Boolean functions in the Boolean representation of the functions \( \delta \) and \( \lambda \),
- \( C = u_1 (N_I + N_S) + u_0 (N_0 + N_S) + \text{overall overhead} \) -- constant for a given assignment length \( N_S \).

So, in the case of the random logic realization, we should minimize the function which increases with the number of product terms \( n_{PT} \), the number of variables in each product term \( n_{VP_j} \), and the number of active product terms \( n_{APT_k} \) for each Boolean function from a Boolean representation of \( \delta \) and \( \lambda \). Additionally, we must remember that the parameters \( n_{PT} \), \( n_{VP} \), and \( n_{APT} \) depend upon each other. If we change the assignment in order to minimize one of them, the Boolean functions representing \( \delta \) and \( \lambda \) will change to the other Boolean functions and, therefore, the other two parameters will be effected too.

On the other hand, even for the minimal assignment length, where \( k \) Boolean variables are required to describe \( K \) internal states and \( 2^{k-1} < K \leq 2^k \), there exist \((2^k)!/((2^k-K)!)) \) ways to assign
the $2^k$ combinations of values of state variables to $K$ states. For $K=9$, there are 10 million versions of the state assignment.

For the greater number of states, the minimization of Boolean representations of $\delta$ and $\lambda$ by comparing the results for all possible versions of the state assignment is impractical or impossible, even using a computer.

Our aim was to find a practical method of state assignment for medium and large sequential machines (machines with more than 10 internal states).

So, we had to find methods that allow us to estimate the complexity of the implementation of a sequential machine, before performing the state assignment and minimizing the Boolean representations of $\delta$ and $\lambda$ functions, i.e. based only on the information given by the next-state and output tables before an assignment.

Based on such approximate estimations of complexity, it is possible to construct a limited in size set of near optimal assignments and then to select the best of these assignments using some real area-oriented implementation-dependent cost functions as a selection criterion.

The selection using the real-cost function has many advantages compared to the selection based on the approximate estimation of complexity. The most important advantages are the following:
1. The real-cost function is considered and not only the estimate of the cost;
2. Various implementation strategies can be considered and the best of them can be selected;
3. Various types of flip-flops for realizing the state memory and real excitation functions can be considered and the best types of flip-flops can be selected.

So, our method for the minimal realization of sequential machines has the following general form:
1. Calculate a limited in size set of near optimal assignments based on an estimation of complexity which is easy enough to calculate from the next-state and output tables;
2. Calculate the minimal Boolean representations of the excitation and output functions for each of the assignments obtained in point (1) for various possible types of flip-flops
realizing the state memory;

3. Calculate, for various possible implementation strategies, the real area-oriented cost function for each solution obtained in point (2) (taking into account not only the real-cost of the combinational part, but also the real-cost of the memory elements of different types);

4. Choose the implementation strategy and the solution minimizing the real-cost functions.

In the method, the sets of near optimal assignments and the minimal solutions are selected for the not necessarily minimum sequential machines. Practical examples show that, in some cases, the implementations of sequential machines with the minimum number of internal states are not minimal and the minimum is achieved for a machine with only partially minimized states. Such a situation occurs quite often, if the full minimization of states does not decrease the number of state variables (the number of excitation functions and the number of flip-flops) in relation to the partial minimization. In the method, the two traditionally independent steps - state minimization and state assignment - are joined into a single process of state assignment and partial state minimization.

For computations performed at the points (2)-(4), well-known algorithms exist or they are quite simple to construct; therefore, we concentrated our attention on point (1) of the method.
3. Partitions and the state assignment.

Let $S$ be any set of elements.

**DEFINITION 3.1** Partition $\pi$ on $S$ is defined as follows:

$$\pi = \{ B_i \mid B_i \subseteq S \text{ and } B_i \cap B_j = 0 \text{ for } i \neq j \text{ and } \cup_i B_i = S \},$$

i.e. a partition $\pi$ on $S$ is a set of disjoint subsets of $S$ whose set union is $S$.

For a given $s \in S$, the block of a partition $\pi$ containing $s$ is denoted as $[s]_{\pi}$ and we will write $[s]_{\pi} = [t]_{\pi}$ to denote that $s$ and $t$ are in the same block of $\pi$. Similarly, the block of a partition $\pi$ containing $S'$, where $S' \subseteq S$, is denoted by $[S']_{\pi}$.

The partition containing only one element of $S$ in each block is called a **zero partition** and denoted by $\pi_s(0)$. The partition containing all the elements of $S$ in one block is called a **one partition** and is denoted by $\pi_s(1)$.

Let $\pi_1$ and $\pi_2$ be two partitions on $S$.

**DEFINITION 3.2** Partition product $\pi_1 \cdot \pi_2$ is the partition on $S$ such that $[s]_{\pi_1 \cdot \pi_2} = [t]_{\pi_1 \cdot \pi_2}$ if and only if $[s]_{\pi_1} = [t]_{\pi_1}$ and $[s]_{\pi_2} = [t]_{\pi_2}$.

**DEFINITION 3.3** Partition sum $\pi_1 + \pi_2$ is the partition on $S$ such that $[s]_{\pi_1 + \pi_2} = [t]_{\pi_1 + \pi_2}$ if and only if a sequence: $s = s_0, s_1, \ldots, s_n = t$, $s_i \in S$ for $i = 1 \ldots n$, exists for which either $[s_1]_{\pi_1} = [s_{i+1}]_{\pi_1}$ or $[s_1]_{\pi_2} = [s_{i+1}]_{\pi_2}$, $0 \leq i \leq n-1$.

From the above definitions, it follows that the blocks of $\pi_1 \cdot \pi_2$ are obtained by intersecting the blocks of $\pi_1$ and $\pi_2$, while the blocks of $\pi_1 + \pi_2$ are obtained by uniting all those blocks of $\pi_1$ and $\pi_2$ which contain common elements.

**DEFINITION 3.4** $\pi_2$ is greater than or equal to $\pi_1$: $\pi_1 \leq \pi_2$ if and only if each block of $\pi_1$ is included in a block of $\pi_2$.

Thus $\pi_1 \leq \pi_2$ if and only if $\pi_1 \cdot \pi_2 = \pi_1$ if and only if $\pi_1 + \pi_2 = \pi_2$.

Let $S_\pi$ be the set of all partitions on $S$. Since the relation "$\leq$" is a relation of partial ordering (i.e. it is reflexive, antisymmetric and transitive), $(S_\pi, \leq)$ is a partially ordered set.

Let $(Z, \leq)$ be a partially ordered set and $T$ be a subset of $Z$. 
DEFINITION 3.5 \( z, z \in \mathbb{Z} \), is the least upper bound (LUB) of \( T \) if and only if:

(i) \( \forall t \in T: z \geq t \),
(ii) \( \forall t \in T: \text{if } z' \geq t \text{ then } z' \geq z \).

\( z, z \in \mathbb{Z} \), is the greatest lower bound (GLB) of \( T \) if and only if:

(i) \( \forall t \in T: z \leq t \),
(ii) \( \forall t \in T: \text{if } z' \leq t \text{ then } z' \leq z \).

DEFINITION 3.6 A partially ordered set \( L = (\mathbb{Z}, \leq) \), which has a LUB and a GLB for every pair of elements, is called a lattice.

It is evident that the set of all partitions on \( S \) together with the relation of a partial ordering \( \leq \), form a lattice with \( \text{GLB}(\pi_1, \pi_2) = \pi_1 \cdot \pi_2 \) and \( \text{LUB}(\pi_1, \pi_2) = \pi_1 + \pi_2 \).

Let \( \pi \) be a partition on the set \( S \). For a given set \( s \in S \), the block of a partition \( \pi \) containing \( s \) is denoted by \([s]_\pi \) and \([s]_\pi = [t]_\pi \) is written to denote that \( s \) and \( t \) are in the same block of \( \pi \), for \( s, t \in S \).

DEFINITION 3.7 A partition \( \pi \) on the set of states \( S \) of the machine \( M = (S, I, O, \delta, \lambda) \) has the substitution property (SP) if and only if \([s]_\pi = [t]_\pi \) implies that \( \forall x \in I: [\delta(s, x)]_\pi = [\delta(t, x)]_\pi \).

DEFINITION 3.8 A partition \( \pi \) on the set of state \( S \) of the machine \( M \) is output consistent if and only if

\([s]_\pi = [t]_\pi \) implies \( \lambda(s) = \lambda(t) \) (a Moore case)
or

\( \forall x \in I \lambda(s, x) = \lambda(t, x) \) (a Mealy case).

In nonminimal sequential machines, compatible states exist. Any two or more compatible states can be replaced by one state without changing the input-output behaviour of a machine.

DEFINITION 3.9 Two states \( s, t \in S \) are compatible if and only if \( \forall x \in I: (\lambda(s, x) = \lambda(t, x) \text{ or } \lambda(s, x) = "-" \text{ or } \lambda(t, x) = "-") \) and \((\delta(s, x) = \delta(t, x) \text{ or } \delta(s, x) = "-" \text{ or } \delta(t, x) = "-") \text{ or } \delta(s, x) \text{ and } \delta(t, x) \text{ are compatible).}
If the compatibility of two states $s$ and $t$ can be reached for $\delta(s,x) = \delta(t,x)$ or $\delta(s,x) = -$ or $\delta(t,x) = -$ for all $x \in I$ then the compatibility is called unconditional. If, at least for one $x \in I$, $\delta(s,x)$ and $\delta(t,x)$ must be two compatible states in order to reach the compatibility of $s$ and $t$, then, their compatibility is called conditional and the condition: "$\delta(s,x)$ and $\delta(t,x)$ must be two compatible states" is called the compatibility condition.

When $s$ and $t$ are compatible, we will write $s \rightarrow t$.

It is quite easy to prove that: a SP-partition $\pi$ on the set of states $S$ of $M$ is output consistent if and only if $[s] \pi = [t] \pi$ implies that $s$ and $t$ are compatible.

**DEFINITION 3.10** A Partial partition $\tau$ on $S$ is a partition $\pi$ on $S'$: $S' \subseteq S$, i.e. a partial partition $\tau$ on $S$ is a set of disjoint subsets of $S$ whose set union is $S'$ being a subset of $S$.

Let $\{S_i\}$ be a set of Boolean variables that represent the states of machine $M$. Let $k$ be the assignment length, i.e. the number of variables in $\{S_i\}$.

The values of a variable $S_j$, $S_j \in \{S_i\}$ for different states from the set $S$ of $M$ (i.e. the values of a binary memory element for different states of $M$) can be described by a partition $\tau_j$ on $S$ only when $\tau_j$ is a two-block partition.

Then, for the states from one block of $\tau_j$: $S_j = 0$ and for the states from the second block $\tau_j$: $S_j = 1$.

If $k$ partitions $\{\tau_j\}$ are used to describe the values of state variables $\{S_i\}$ and combinations of the values of the state variables represent the states of $M$ unambiguously, then, these $k$ partitions have to distinguish each element of the set $S$ from each other element of this set. So, the set of partitions $\{\tau_j\}$ used for the state assignment has to be an orthogonal family of partitions, i.e. $\tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_k = \tau(0)$.

In order to fulfill this condition, the number of elements in each block of each partition from the set $\{\tau_j\}$ does not exceed $2^{k-1}$.

**DEFINITION 3.11** A partition $\tau$ on $S$ is called a proper partition for a given assignment length $k$ if and only if the two following conditions are satisfied:

(i) the number of blocks in $\tau_j$ is equal 2;
(ii) the number of elements in each block is less than or equal to $2^{k-1}$.

For sequential machines with the minimum number of internal states, only the proper partitions are candidates for the state assignment; all other partitions are useless.

For non-minimal machines, each two-block partition, not necessarily proper, can be considered as a candidate for the state assignment.

In this case, the orthogonality condition is replaced with two other conditions: the separation condition and the closure condition.

The separation condition states that each pair of a machine's incompatible states must be separated by at least one partition from the set of $k$-partitions $\{\tau_j\}$ that are used for the state assignment in order to distinguish two incompatible states in the code space. It is not necessary to distinguish the compatible states. They can be assigned with the same codes and, in this way, they will be mapped into one state of a machine implementing a certain machine M. On the other hand, it is not necessary to perform the full minimization of states for a machine M because, in some cases, the partial minimization will lead to an optimal implementation. In each case, however, the closure condition must be satisfied, this states that for each block of the partition $\pi$, which is the product of all partitions $\{\tau_j\}$ used for the state assignment, for each $x \in I$, the next-states for the states contained in a given block of $\pi$ must be contained in one block of $\pi$ and the output values for the states contained in a given block of $\pi$ must be identical or some of them can be "don't care" values. In other words, the closure condition states that the partition $\pi$, being the product of all partitions $\{\tau_j\}$ used for state assignment, must be an output consistent SP-partition.

Every set of two-block partitions which satisfies the separation condition and the closure condition can be used for the state assignment and it will be referred to as the final family of partitions (FFP).

For the case of a minimal machine, it is evident that only the separation condition is active, because all states are
incompatible, and because each state must be separated from the others in at least one partition. The product of partitions used to state assignment has to be a zero partition \(\pi(\emptyset)\). \(\pi(\emptyset)\) is a trivial output consistent SP-partition. So, in the case of a minimal machine, any orthogonal family of proper partitions is a FPP.

Now, the problem of the optimal state assignment can be formulated in the following manner:
find such two-block partitions on the set of states \(S\) of machine \(M\) which form a final family of partitions (FFP) that minimizes the cost function of the implementation of \(M\).

In the practical algorithm, first we generate a limited in size set of near optimal assignments, i.e. a limited set of near optimal FFP's; then, a choice is made from this set of the FFP which minimizes the real-cost function, taking into account various possible types of flip-flops and various possible implementation strategies.

Now pay more attention to the construction of a limited set of near optimal FFP's (SNOFFP).

The greatest assignment length \(k\) for a given machine \(M\) is always known: it is the assignment length for the realization: "one state - one flip-flop" (this assignment length equals \(|S|\), where \(|S|\) is a number of states of \(M\)).

The least assignment length \(k\) is known for minimal machines (it is the assignment length \(k\) which satisfies the inequality \(2^{k-1} \leq |S| \leq 2^k\)) but, in general, it is unknown a priori for unminimal machines. In the latter case, the least possible assignment length \(k\) satisfies the inequality \(2^{k-1} \leq |S_{\min}| \leq 2^k\), but \(|S_{\min}|\), a number of states in a minimal machine equivalent to \(M\), is unknown prior to minimization. Only in some simpler cases, when the number of pairs of compatible states is small, can \(S_{\min}\) be easily estimated using the knowledge of these pairs. On the other hand, practical examples show, that, in most cases, the implementations of sequential machines with the least assignment length \(k\) are optimal, while implementations with the values of \(k\) higher than minimal are only sometimes optimal.

So, it is necessary to concentrate on assignments with the
minimal or near minimal length first and then maybe to consider
the longer assignments.

Let \( k_{\text{min}} \) be the least possible assignment length. Let \( k^{n_{\text{min}}} \) be
the assignment length satisfying the inequality:
\[
2^{k^{n_{\text{min}}}-1} < |S| < 2^{k^{n_{\text{min}}}},
\]
where \( |S| \) is a number of states in
a nonminimal machine.
Let SNOFFP(k) be a limited set of near optimal final families of
partitions for an assignment length \( k \).
Let CE\(_{\text{min}}\)(k) be a minimal value of the complexity estimate reached
for FFP's from SNOFFP(k).
Let CE\(_{\text{min}}\) be a minimal value of the complexity estimate reached
for FFP's from all considered SNOFFP(k)'s.

Now, the algorithm for constructing a limited set of near
optimal final families of partitions (SNOFFP) can be formulated
in the following manner:

1. SNOFFP = \( \emptyset \).
2. If \( k_{\text{min}} \) is known then \( k := k_{\text{min}} \)
   else \( k := k^{n_{\text{min}}} \)
3. Find SNOFFP(k); calculate CE\(_{\text{min}}\)(k);
   if CE\(_{\text{min}}\)(k) < CE\(_{\text{min}}\) then CE\(_{\text{min}}\) := CE\(_{\text{min}}\)(k).
4. If \( k_{\text{min}} \) is known then
   if CE\(_{\text{min}}\)(k) ≤ CE\(_{\text{min}}\) and \( k < |S| \) then
   begin SNOFFP := SNOFFP ∪ SNOFFP(k);
   \( k := k+1 \); go to 3 end else go to 6.
5. If \( k_{\text{min}} \) is unknown then
   if \( k \leq k^{n_{\text{min}}} \) then
   if SNOFFP(k) ≠ \( \emptyset \) then
   begin
   SNOFFP := SNOFFP ∪ SNOFFP(k);
   \( k := k-1 \); go to 3 end
   else
   begin \( k := k^{n_{\text{min}}}+1 \); go to 3 end;
   else
   if CE\(_{\text{min}}\)(k) ≤ CE\(_{\text{min}}\) and \( k < |S| \) then
   begin
   SNOFFP := SNOFFP ∪ SNOFFP(k);
   \( k := k+1 \); go to 3
end
else go to 6

6. SNOFFP contains the limited number of near optimal final families of partitions; STOP.

In steps 4 and 5 of the algorithm, the condition \( CE_{\text{min}}(k) \leq CE_{\text{min}} \) can be replaced in a more sophisticated version of the algorithm, with the condition:

\[
CE_{\text{min}}(k) \leq CE_{\text{min}} \quad \text{or} \quad \\
CE_{\text{min}}(k-1) \leq CE_{\text{min}} \quad \text{or} \\
\ldots \quad \ldots \quad \text{or} \\
CE_{\text{min}}(k-n) \leq CE_{\text{min}},
\]

for some \( n \),

i.e. the higher values of the assignment length \( k \) have not to be considered if not only for one, the last considered, value of \( k \) but for a number of last considered values of \( k \), the minimal value of the complexity estimation \( CE_{\text{min}}(k) \) is greater than \( CE_{\text{min}} \).

The most important step of the algorithm above is to find SNOFFP(\( k \)), i.e. to solve the problem:

\textit{for a given assignment length \( k \), find a limited set of near optimal final families of partitions on the set of states \( S \) of a sequential machine \( M \), based on an approximated estimate of implementation complexity, which is easy enough to calculate from the next-state and output tables.}

In the next chapters, a method will be introduced which allows us to solve such a formulated problem of constructing the limited in size set of near optimal assignments.
4. The method of maximal adjacencies.

The method of maximal adjacencies allows us to construct, for a given assignment length $k$, a limited set of near optimal state assignments.

From Chapter 2, it is known for a PLA implementation that the state assignment should minimize the number of product terms in the Boolean representations of the next-state and output functions $\delta$ and $\lambda$ in order to be optimal for a given assignment length $k$.

It is known also for a random logic implementation, that the complicated function, which grows with the total number of product terms, the number of variables in each product term and the number of active product terms in each function from the Boolean representation of $\delta$ and $\lambda$, has to be minimized.

The problem is the inability to analyze the Boolean representations of functions $\delta$ and $\lambda$ after the assignment and to carry out minimization for all the possible state assignments.

The number of product terms and the other two parameters (in the case of random logic) have to be estimated before making the state assignment and minimization of the resulting Boolean representation of $\delta$ and $\lambda$, based only on the knowledge of the next-state and output tables.

The minimal number of product terms, which in a logical sum represent a given Boolean function, depends on the number of full product terms (product terms of the order zero, 0-cubes) for a given function (i.e. the number of "1"s in a Karnaugh map representing a given function) and of the adjacency of these full product terms (i.e. adjacency of "1"s in a Karnaugh map representing a given function).

Two binary sequences (e.g. two input codes, two state codes, two product terms) $a$ and $b$ of the same length $l$ are adjacent if the Hamming distance between them is equal one, where the Hamming distance between $a$ and $b$ is the number of positions where they differ.

The adjacency of the full product terms represents the possibility of constructing the product terms of higher orders for a given function. In order to obtain the minimal "sum of products" representation of a Boolean function, a minimal set of terms has to be chosen from the set of all possible product terms.
of a function, which covers all the full product terms (i.e. these terms, in a logical sum, represent a given function). If, for a given number of full product terms, the number of adjacent pairs of full product terms is higher, then, the chance of creating larger groups of adjacent full product terms, i.e. the product terms of higher orders, will be higher too, because the pairs of adjacent full product terms (i.e. product terms of the order 1) will be forced together, if there are enough of them.

Each product term of a higher order covers a number of full product terms and the number of covered full product terms grows with the order of a given higher order product term.

So, having more adjacent full product terms, there is a chance of covering all the full product terms by a smaller number of larger product terms, i.e. a chance to obtain fewer product terms which in a logical sum represent a given Boolean function and fewer variables in some of those product terms.

It is evident that the adjacency of "1"s (i.e. adjacency of full product terms) of a Boolean function and the adjacency of "0"s are coupled together.

If, for a given number of "1"s and "0"s of a Boolean function the adjacency level of "0"s increases, then, the adjacency level of "1"s increases too (or at least does not decrease - in the case of an incompletely specified Boolean function) and vice versa.

Therefore, the adjacency level of "0"s of a Boolean function has the same influence on creating the higher order product terms as the adjacency level of "1"s.

So, in order to find near optimal state assignments, look for those final families of partitions, where the Boolean functions that result from using these families for assignment, describe the values of the next-state and output variables of a sequential machine with a large number of adjacent "1"s and "0"s.

Based on this observation, a method for constructing the set of near optimal final families of partitions has been developed. The method creates, for a given assignment length k, a set of final families of partitions that maximize the adjacency level of "1"s and "0"s of Boolean functions obtained with a given family used to state assignment. Calculations are based only on the information from the next-state and output tables. Firstly, the adjacency conditions for input-states, present-states--next-states and state-outputs dependencies are determined.
This is the only step where the access to the machine tables is required. Then, these three sorts of adjacency conditions are combined together and ordered, according to the offered level of adjacency, forming the ordered list of adjacency conditions.

The first two steps of the method include simple calculations that are very fast and need little memory.

The third step consists of creating the final families of partitions based on the ordered list of adjacency conditions.

In this step, the adjacency conditions are considered in the order of their ordered list and the final families of partitions are created that satisfy the greatest number of compatible adjacency conditions.

4.1 The adjacency conditions for input-state dependencies.

As opposed to the codes for states, which have to be chosen, the codes for inputs and outputs of a sequential machine are usually predefined, because inputs in most cases are formed in most cases with direct signals from the surroundings of the machine, and outputs are direct control signals sent by the machine to its surroundings.

With regard to the input codes, two cases are considered here. In the first case, all code combinations from the code space for inputs are used in order to encode the input symbols. In the second case, there are unused input code combinations.

Now, taking the first case:

If, for a given present state $s_1$ and for two codewise adjacent input symbols (values) $x_j$ and $x_k$, the next-states $s_m = \delta(s_1, x_j)$ and $s_n = \delta(s_1, x_k)$ are placed in one block of the partition $\tau_1$ on $S$ used for state assignment, then in the Boolean function describing values of the state variable $S_1$ assigned according to $\tau_1$, two adjacent "1"s or two adjacent "0"s are obtained depending upon whether the block of $\tau_1$ containing states $s_m$ and $s_n$ is assigned with "1" or "0".

So, the pairs of states $s_m$ and $s_n$, where $s_m = \delta(s_1, x_j)$ and $s_n = \delta(s_1, x_k)$, represent conditions for obtaining adjacent ones or zeroes in the Boolean next-state functions for a given present-state $s_1$ and two adjacent input combinations $x_j$, $x_k$.

Based on the next-state table, a simple computation can find
the adjacency conditions for all $s_1: s_1 \in S$ and all pairs $(x_j, x_k)$ of adjacent inputs (the greatest number of possible adjacency conditions is smaller than or equal to $|S| \cdot (|S|-1)/2$). For each of these adjacency conditions, the total number of times a given condition occurs in the whole next-state table of a machine is calculated (i.e. for all $s_1: s_1 \in S$ and all adjacent pairs of inputs $(x_j, x_k): x_j, x_k \in I$). We denote that two input symbols $x_j, x_k$ are adjacent, by writing $x_j \mid x_k$.

When not all the input code combinations are used for expressing the input symbols, the input symbols coded with adjacent combinations (in the sense of a Gray code) must be considered, but the notion of "adjacency" of the two input symbols has to be extended for the case when there is a subspace of the input code space containing the two input symbols alone (not containing any other input symbol).

Similarly, as for all the adjacent pairs of inputs, for all the extended adjacent pairs of inputs the adjacency conditions have to be calculated.

For each calculated condition, the dimension of the extended subspace containing a given pair of extended adjacent states should be noted and, then, the total of subspace dimensions for the whole next-state table should be calculated for each condition.
Example 4.1

Consider a sequential machine with the following next-state table where all the input code combinations are used.

<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Next-state table

<table>
<thead>
<tr>
<th>adjacency condition</th>
<th>number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,4</td>
<td>4</td>
</tr>
<tr>
<td>1,4</td>
<td>4</td>
</tr>
<tr>
<td>2,3</td>
<td>4</td>
</tr>
<tr>
<td>1,3</td>
<td>4</td>
</tr>
</tbody>
</table>

List of adjacency conditions
Example 4.2

Consider the sequential machine with the following next-state table where some input code combinations are unused.

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>000</th>
<th>001</th>
<th>011</th>
<th>010</th>
<th>110</th>
<th>111</th>
<th>101</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Next-state table

<table>
<thead>
<tr>
<th>x₀</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
<th>x₅</th>
<th>x₆</th>
<th>x₇</th>
<th>x₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x₃</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>x₆</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Map of 4 used input symbols in a 3-dimensional input code space

<table>
<thead>
<tr>
<th>number of subspace dimensions</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀</td>
<td>x₁</td>
<td>x₀</td>
<td>x₁</td>
<td>x₀</td>
</tr>
<tr>
<td>1</td>
<td>2,3</td>
<td>3,4</td>
<td>1,2</td>
<td>1,4</td>
</tr>
<tr>
<td>2</td>
<td>1,4</td>
<td>1,2</td>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3</td>
<td>1,3</td>
<td>3,4</td>
<td>1,2</td>
<td>2,4</td>
</tr>
<tr>
<td>4</td>
<td>1,3</td>
<td>1,2</td>
<td>3,4</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Table of adjacency conditions

- x₀|x₁, x₁|x₂ - adjacent pairs
- x₀|x₆, x₃|x₆ - extended adjacent pairs

<table>
<thead>
<tr>
<th>adjacency</th>
<th>adjacency weight*</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,3</td>
<td>3</td>
</tr>
<tr>
<td>3,4</td>
<td>4</td>
</tr>
<tr>
<td>1,2</td>
<td>6</td>
</tr>
<tr>
<td>1,3</td>
<td>4</td>
</tr>
<tr>
<td>1,4</td>
<td>3</td>
</tr>
<tr>
<td>2,4</td>
<td>4</td>
</tr>
</tbody>
</table>

List of adjacency conditions

(* number of occurrences and/or total of subspace dimensions)
4.2 The adjacency conditions for present-state — next-state dependencies.

If, for a given value of input $x_j$, the states $s_k$ and $s_1$ are assigned to codewise adjacent combinations of state variables and the next-states $s_m = \delta(s_k, x_j)$ and $s_n = \delta(s_1, x_j)$ are placed in one block of the partition $\tau_i$ on $S$ used for state assignment, then, in the Boolean function describing values of the state variable $S_i$ assigned according to $\tau_i$ two adjacent "1"s or two adjacent "0"s are obtained depending on whether the block of $\tau_i$ containing states $s_m$ and $s_n$ is assigned with "1" or "0".

So, conditions of the type: "if two states $s_k$ and $s_1$ are codewise adjacent and two next-states $s_m = \delta(s_k, x_j)$ and $s_n = \delta(s_1, x_j)$ are placed in one block of partition $\tau_i$ used for state assignment" will constitute the conditions for obtaining adjacent ones or zeroes in the Boolean representation of the next-state function for a given input value $x_j$ and two codewise adjacent states $s_k$ and $s_1$.

Based on the next-state table, these adjacency conditions can be calculated simply for all possible pairs of different states $s_k$ and $s_1$: $s_k, s_1 \in S$ and for all input values $x_j$. The number of all such conditions is equal to $(|S| \cdot (|S| - 1)/2) \cdot |I|$. In this way, a table of adjacency conditions is made.

Each entry of this table, for a given pair $s_k|s_1$ and a given $x_j$, contains the pair of states $s_m = \delta(s_k, x_j)$ and $s_n = \delta(s_1, x_j)$, if $s_m \neq s_n$ or it contains "don't care", if $s_m = s_n$ or $s_n = "don't care"$ or $s_n = "don't care"$.

For each pair of states $(s_m, s_n)$ from the adjacency condition for a given pair of adjacent states $s_k|s_1$, the number of times this pair $(s_m, s_n)$ occurs in the table of adjacency conditions, for a given $s_k|s_1$ and all used input symbols $x_j: x_j \in I$, is calculated, as well as, the number of "don't care" entries in the table of adjacency conditions for a given $s_k|s_1$ and all used input symbols $x_j$.

In this way, the list of adjacency conditions is prepared.
Consider the same next-state table as in Example 4.1.

<table>
<thead>
<tr>
<th>$s_k$</th>
<th>$s_1$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2,4</td>
<td>2,4</td>
<td>1,3</td>
<td>2,4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1,2</td>
<td>—</td>
<td>—</td>
<td>3,4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2,3</td>
<td>2,4</td>
<td>1,3</td>
<td>1,4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,4</td>
<td>2,4</td>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3,4</td>
<td>—</td>
<td>—</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1,3</td>
<td>2,4</td>
<td>1,3</td>
<td>1,3</td>
</tr>
</tbody>
</table>

**Table of adjacency conditions**

When the list of adjacency conditions has been constructed, it has to be checked in order to discover the pairs of states ($s_m, s_n$) which in a given condition have to be adjacent and, simultaneously, have to be contained in one block of a partition $r_i$ used for state assignment. If two given states $s_m$ and $s_n$ are adjacent codewise, they will be contained in the same block in exactly $(k-1)$ from $k$ partitions used for state assignment and they have to be in two different blocks in exactly one partition. So, prior to constructing the final family of partitions, it is known that when the given states $s_m$ and $s_n$ are adjacent, i.e. $s_m | s_n$, then, the pair of states $(s_m, s_n)$ will be contained in one block in exactly $(k-1)$ partitions. Therefore, the total number of adjacencies obtained for a given pair $(s_m, s_n)$ will be equal to $(k-1) \cdot sa(m,n)$, where $sa(m,n)$ is the number of adjacencies obtained for the pair $(s_m, s_n)$ in the adjacency condition for $s_m | s_n$. So, for
the adjacency condition with $s_m|s_n$, $sa_{(m,n)}$ adjacencies are obtained independently of the conditions for putting other pairs of states into one block of partitions. Similarly, if the adjacency condition for a given $s_k|s_1$ contains "don't cares", then, the $k\cdot nd$ adjacencies are obtained independently, where $nd$ is a number of "don't cares". Altogether, for a given $s_m|s_n$:

$$sa = (k-1)\cdot sa_{(m,n)} + k\cdot nd$$

adjacencies are obtained independently of the conditions for putting other pairs of states into one block of partitions.

The list of adjacency conditions is converted by calculating and noting the number of adjacencies obtained for each pair $s_k|s_1$ independently of the conditions for putting other pairs of states into one block of partitions and by removing pairs $(s_k,s_1)$ which, in a given condition, have to be adjacent and have to be put into one block of a partition $\tau_1$ that will be used for a state assignment.
Example 4.4

Convert the list of adjacency conditions from example 4.3. Let $k=2$.

<table>
<thead>
<tr>
<th>pair of adjacent states</th>
<th>pairs of next-states and number of their occurrences</th>
<th>number of independently reached adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2,4 1,3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td></td>
<td>3 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1,2 3,4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1,3 2,3 2,4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3 1,4 2,4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,2 3,4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1,3 2,4</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td></td>
<td>3 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Converted list of adjacency conditions

4.3 The adjacency conditions for state-output dependencies.

From the point of view of dependencies of the output function $1$ the following two types of sequential machines are distinguished:

1. Moore machines - where outputs depend only on states (i.e. $1: S \rightarrow 0$).
2. Mealy machines - where outputs depend on states as well as inputs (i.e. $1: S \times I \rightarrow 0$).

Assigning states requires these two types of sequential machines to be treated in two different ways: especially, if the outputs depend substantially on the inputs, in the case of a Mealy machine.

In the case of Moore machines, firstly, try to use for state
assignment the two block state partitions obtained from the S-O partition pairs (output induced state partitions) and then take into account the other dependencies. This is necessary because implementation of the output function \( \lambda \) is greatly simplified if a set of state partitions \( \{ \tau_i \} \) forming S-O pairs with output partitions \( \{ \omega_k \} \) is used for the state assignment. Assigning a state variable \( s_i \) according to the partition \( \tau_i \) which forms with \( \omega_k \) a S-O pair, means that values of the output variable \( y_k \) which introduces the partition \( \omega_k \) on the output set \( O \), are the same as the values of the state variable \( s_i \) or the negated values of \( s_i \). So, the implementation of the Boolean output function \( \lambda_k \) for the output variable \( y_k \), in this case, is very simple and cost-free.

In the case of a Mealy machine, such a simplification of the realization of the output function \( \lambda \) is not possible, because outputs will depend not only on the states but on the inputs too. In this case, state partitions from the S-O pairs or the output induced state partitions can still be useful but they are less important than for a Moore machine.

Therefore, the extension of the adjacency concept to dependencies of the outputs of states is especially interesting for a Mealy machine.

If the states \( s_k \) and \( s_1 \) are adjacent codewise, then:
- for a given input value \( x_j \), two adjacent "1"s or two adjacent "0"s are obtained in the Boolean function which describes the value of the output variable \( y_i \) if and only if \( \lambda_i(s_k,x_j) = \lambda_i(s_1,x_j) \) or \( \lambda_i(s_k,x_j) = \lambda_i(s_1,x_j) \) or \( \lambda_i(s_k,x_j) = \lambda_i(s_1,x_j) \);
- for all possible values \( x_j : x_j \in I \), as many adjacencies of ones and zeroes are obtained as many times the above condition is satisfied.

So, in the case of state-output dependencies, the conditions for obtaining adjacent ones or zeroes have the form: "if two states \( s_k \) and \( s_1 \) are adjacent, then, in the Boolean function \( \lambda_i \) describing values of the output variable \( y_i \), \( n \) adjacencies exist".

By computing the total number of dependencies for each pair of states \( s_k \mid s_1 \): \( s_k,s_1 \in S \) and for all output variables \( y_i \), and summing the number of adjacencies obtained for a given \( s_k \mid s_1 \) for each \( y_i \), the list of adjacency conditions is obtained.
Example 4.5

Consider the following output table:

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>01</td>
</tr>
</tbody>
</table>

Next-state table

<table>
<thead>
<tr>
<th>s_k</th>
<th>s_1</th>
<th>y_1</th>
<th>y_2</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

List of adjacency conditions

4.4 Combined adjacency conditions.

It has been stated already that, in order to obtain near optimal state assignments, sets of the final families of partitions should be created which maximize the adjacency level of "1"s and "0"s of the Boolean functions resulting from the use of a given family to state assignment.

In the last three paragraphs, three sorts of adjacency conditions were considered, following from input-states, present-state--next-state and state-output dependencies. If a given adjacency condition is satisfied, then, in a Boolean representation of the next-state or output function, a given number of adjacent "1"s or "0"s will be obtained. It has been shown how to create the lists for these three sorts of adjacency conditions, based on the information given in the machine tables.

The conditions for each of these three sorts of adjacency conditions are equally important for finding near optimal assignments. Not the sort of a condition, but the guaranteed number of adjacencies and the possibility to combine with other conditions decides about the importance of a given condition.

Therefore, all three sorts of adjacency conditions should be treated equally.
final families of partitions that maximize the adjacency level of "1"s and "0"s based on adjacency conditions, it will be shown how to obtain one list of combined adjacency conditions from three lists of conditions for three different conditions.

The adjacency conditions for input-state dependencies have the form:
"if a pair of states \( (s_m, s_n) \) is contained in one block of a partition \( \tau_1 \) that is used for a state assignment, then, \( i_a \) adjacencies will be obtained".

The adjacency conditions for present-state--next-state dependencies are of the form:
"if two states \( s_k | s_1 \) are adjacent codewise then \( s_a \) adjacencies are obtained and, additionally, if each pair of states \( (s_m, s_n) \), such that \( s_m = \delta(s_k, x_j), s_n = \delta(s_1, x_j); j=1,\ldots,|\Gamma|; \) is placed in one block of a partition \( \tau_1 \) that is used for a state assignment, then, for each such a pair, \( s_a(m,n) \) adjacencies will be obtained".

The adjacency conditions for state-output dependencies have the form:
"if two states \( s_k | s_1 \) are adjacent codewise, then, in Boolean functions implementing the output function \( \lambda \), \( s_o \) adjacencies will be obtained".

The adjacency conditions for present-state--next-state dependencies are most complicated; they cannot be converted into one of the simpler forms of conditions for input-state or state-output dependencies. Therefore, combining the conditions for the three types considered, in order to form a single combined condition, a condition is created that has the same form as the condition for the present-state--next-state dependencies.

Consider the adjacency condition for present-state--next-state dependencies of a given pair of adjacent states \( s_k | s_1 \).

The adjacency condition for state-output dependencies of the same pair of adjacent states \( s_k | s_1 \) can be combined directly with the condition considered. In order to do that, it is only necessary to add to the number of adjacencies \( s_o \) obtained for present-state--next-state dependencies, the number of
adjacencies $oa$ obtained for the state-output dependencies.

The adjacency condition for input-state dependencies of a given pair of states $(s_k, s_1)$ can be combined with the considered condition too. When $s_k|s_1$ is considered, the pair of states $(s_k, s_1)$ is contained in one block of exactly $(k-1)$ partitions $T_1$ that are used for the state assignment and it is split into exactly one partition. So, it is necessary to add the number of adjacencies $ia$ obtained from input-state dependencies multiplied by $(k-1)$ to the number of adjacencies $sa$ obtained from present-state--next-state dependencies.

In order to combine the adjacency condition for the state-output dependencies of the same pair of states $s_k|s_1$ and the adjacency condition for the input-states dependencies of the same pair of states $(s_k, s_1)$ with the adjacency condition for present-state--next-state dependencies of a given pair of states $s_k|s_1$, a new value of the number of adjacencies $na$ must be calculated for a given pair of states $s_k|s_1$ (treated as adjacent codewise states), where:

$$na = sa + oa + (k-1) \cdot ia.$$

However, the adjacency condition for the input-state dependencies of a given pair of states $(s_k, s_1)$ should be combined not only with the adjacency condition for the present-state--next-state dependencies of a given pair of states $s_k|s_1$ but also with all the adjacency conditions for the present-state--next-state dependencies containing a given pair $(s_k, s_1)$ as the pair which has to be included in one block of a partition $T_1$ that is used for a state assignment. In order to do that, the number of adjacencies obtained for this pair $(s_k, s_1)$ for the input-states dependencies $ia_{(k,1)}$, must be simply added to the number of adjacencies obtained for such a pair $(s_k, s_1)$ from the present-state--next-state dependencies $sa_{(k,1)}$; in this way, a new total number of adjacencies $na_{(k,1)} = sa_{(k,1)} + ia_{(k,1)}$ is obtained for a given pair $(s_k, s_1)$. 
**Example 4.6**

Consider the lists of adjacency conditions in Examples 4.1, 4.4 and 4.5 and combine the conditions from these lists in order to obtain a common list of adjacency conditions.

<table>
<thead>
<tr>
<th>pair of adjacent states</th>
<th>pair of next-states and number of their occurrences</th>
<th>number of unconditionally reached adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2,4 1,3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>7 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1,2 3,4</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1,3 2,3 2,4</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5 5 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3 1,4 2,4</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>5 5 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,2 3,4</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1,3 2,4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>7 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Common list of adjacency conditions**

**4.5 Ordering of adjacency conditions.**

When constructing the near optimal final families of partitions, the objective is to find such families for which the total number of "∅"s and "1"s adjacencies is big, i.e. families satisfying as many adjacency conditions as possible and satisfying the conditions where the number of adjacencies is large.

Therefore, before starting to construct the final families of proper partitions, it is a good idea to put the adjacency conditions contained in the common list of conditions in order according to the number of adjacencies obtained when a given condition is satisfied.
Ordering the common list of adjacency conditions, starts from the conditions that have the most adjacencies and for a given number of adjacencies for conditions with the most "don't cares". The number of "don't cares" has not only influence on the number of adjacencies, but also it displays the possibility of combining a given adjacency condition with other conditions.

Based on the ordered list of adjacency conditions, the final families of partitions are constructed by combining the greatest possible number of compatible adjacency conditions in the order defined by the ordered list of conditions.

Making calculations in this order, the near optimal solutions are computed at first and there is a good chance of reaching the best solution among one of the first computed solutions.

When ordering the adjacency conditions, the number of adjacencies obtained unconditionally for a given pair of adjacent codewise states $s_k|s_1$ has to be taken into account, as well as the number of adjacencies reached under the condition that given pairs of states $(s_m,s_n)$ are contained in one block of the partitions used for the state assignments. However, rarely for all pairs $(s_m,s_n)$ associated with a given pair $s_k|s_1$, appropriate conditions can be true simultaneously, especially, if the conditions for the previously considered pairs of adjacent states have to be also satisfied. In practice, satisfying only the conditions that offer many enough adjacencies is really important. Therefore, considering a given adjacency condition, firstly, satisfying the conditions for pairs $(s_m,s_n)$ giving the greatest number of adjacencies should be tried and then, the other conditions, if it is still possible to satisfy them.

So, for a given adjacency condition, the pairs of states $(s_m,s_n)$ have to be put in order according to the number of adjacencies, starting from pairs with the greatest number of adjacencies.

Estimating, for a certain adjacency condition, the number of adjacencies possible to reach upon the condition that some pairs $(s_m,s_n)$ are contained in one block of partitions that will be used to state assignment, we should take into account only the first pairs with the most adjacencies.

The largest considered number of simultaneously satisfied conditions, $MNSC$, is a parameter of the algorithm, and it should be chosen experimentally.
Similarly, the average number of partitions \( T_T \), \( MNP \), containing the pair of states \((s_m, s_n)\) in their blocks, should be selected experimentally.

The first experiments showed, that good estimates of the number of adjacencies were obtained if \( MNSC = k \) or \( MNSC = k-1 \) and \( MNP = \lceil k/2 \rceil \).

From the considerations mentioned above, it follows that the estimation of the total number of adjacencies, \( tna \), for a given adjacency condition can be obtained as follows:

\[
tna = na + MNP \cdot \sum_{i=1}^{MNSC} na_{(m,n)}^i .
\]

**Example 4.7**

Consider the common list of adjacency conditions from the example 4.6.

Let \( MNSC=k=2 \), \( MNP=\lceil k/2 \rceil=1 \), \( tna = na + \sum_{i=1}^{2} na_{(m,n)}^i \).

<table>
<thead>
<tr>
<th>pair of adjacent states</th>
<th>estimation of the total number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Estimations of the total number of adjacencies for an adjacency condition with adjacent states \( s_k | s_1 \).
since, satisfying only the conditions for pairs of states \((s_m,s_n)\) offering a sufficient number of adjacencies is really important from the practical point of view, it is possible to simplify the adjacency conditions by ruling out the pairs \((s_m,s_n)\) with a smaller number of adjacencies.

A given pair \((s_m,s_n)\) can be contained in \((k-1)\) partitions \(r_i\), at most. So, if for a given pair \((s_m,s_n)\) the condition:

\[
\text{MNSC} \cdot (k-1) \cdot n_{a_{(m,n)}} \ll n_a \quad \text{or the condition:} \quad \text{MNSC} \cdot (k-1) \cdot n_{a_{(m,n)}} \ll \left( \sum_{i=1}^{\text{MNSC}} n_{a_{(m,n)}} i \right)
\]

is satisfied, such a pair can be ruled out.
4.6 Constructing the final families of partitions based on the ordered list of adjacency conditions.

In this section, constructing FFP's for minimal machines will be considered. In the next section, it will be shown how the method should be supplemented in order to cover non-minimal machines.

Constructing the near optimal final families of partitions, it is necessary to combine as many adjacency conditions as possible, considering the conditions according to the order given by the ordered list of adjacency conditions.

Making calculations in such an order, starts with the final families of partitions that give a high adjacency level and there is a good chance of reaching the best of them in one of the earliest computation steps.

The maximal possible number of conditions for adjacency of states $s_k|s_1$, which can be fulfilled together, is limited and equal to $\text{MNAC}^{k\times l}$ (maximal number of adjacency conditions for $|S|$ states and an assignment length $k$, which can be fulfilled together).

If all the code combinations are used to state assignment (i.e. $|S| = 2^k$) then $\text{MNAC}^{k\times l}$ is equal to the number of edges in a $k$-dimensional cube, i.e.

$$\text{MNAC}_k = 2 \cdot \text{MNAC}_{k-1} + 2^{k-1} \text{ for } k=1,2,... \text{ and } \text{MNAC}_0 = 0.$$

If some conditions are not used (i.e. $|S|<2^k$), then, $\text{MNAC}^{k\times l}$ is equal to the maximal number of edges between $|S|$ nodes in a $k$-dimensional cube. If $|S|<2^k$, then, it is possible to represent $|S|$ in the form $|S|= a_{k-1}2^{k-1}+a_{k-2}2^{k-2}+...+a_12^1+a_02^0$ (i.e. the binary form with the binary digits $a_{k-1}a_{k-2}...a_1a_0$), where $a_i = 0,1$ for $i = 0,\ldots,k-1$.

Let $\text{MNAC}_i$ be the number of edges in an $i$-dimensional cube.

The maximal number of adjacency conditions that can be fulfilled together, $\text{MNAC}^{k\times l}$, when only some of the code combinations are used, is given by:

$$\text{MNAC}^{k\times l} = \sum_{i=0}^{k-1} a_i \cdot (\text{MNAC}_i + (\sum_{j=i+1}^{k-1} a_j) \cdot 2^i).$$

Combining adjacency conditions in order to form FFP's for minimal machines the following constraints must be taken into account:
(1) If two states \( s_k \parallel s_1 \) have to be adjacent codewise, then, they must be contained in two different blocks of just one two-block partition which is member of a FFP. In all other partitions from the FPP, they must be contained in one block.

(2) Each pair of incompatible states \((s_m, s_n)\) must be separated, in at least one partition, from an FFP (separate condition).

(3) Only the proper partitions are useful for state assignment of a minimal machine and only they can be members of FPP's.

(4) The FPP for a minimal machine is an orthogonal family of proper partitions.

(5) Each state \( s_k : s_k \in S \) may be codewise adjacent with at most \( k \) other states \( s_1 : s_1 \in S \).

(6) The number of adjacency conditions, which can be fulfilled together for a given number of states \(|S|\) and a given assignment length \( k \), is not greater than \( MNAC''_k' \).

Now, it can be shown how a limited in size set of near optimal families of final partitions for an assignment length \( k \), SNOFFP(\( k \)), can be constructed, using a "branch and bound" concept, and considering the adjacency conditions in the order of their ordered list.

Each FFP \( \epsilon \) SNOFFP(\( k \)) must contain \( k \) proper partitions, i.e. \( k \) two-block partitions containing at most \( 2^{k-1} \) elements in each of their blocks.

When starting computations for the \( n \)-th considered adjacency condition, such a set of \( k \)-tuples of partial proper partitions is given, that, each of their \( k \)-tuples satisfies the previously considered \((n-1)\) adjacency conditions. For the condition considered first, this \( k \)-tuple consists of partial proper partitions containing empty blocks (no conditions being satisfied).

For the \( n \)-th adjacency condition considered, a set of all \( k \)-tuples of partial proper partitions has to be constructed in order to satisfy the previously considered \((n-1)\) conditions and the \( n \)-th condition, in the following manner:

(1) For a given pair of states, \( s_k \parallel s_1 \), and for each \( k \)-tuple of the partial proper partitions \( \{\tau_{(n-1)}\} \), construct all \( k \)-tuples of partial proper partitions \( \{\tau'_n\} \) such that: \( \tau_{(n-1)} \leq \tau'_n \) (i.e. all blocks of \( \tau_{(n-1)} \) are contained in blocks of \( \tau'_n \)).
and states, \( s_k \mid s_1 \), are contained in different blocks of only one partition from \( \{ \tau'_1 \} \).

(2) For each pair of states, \((s_m, s_n)\), and for each \( n \)-tuple of the partial proper partitions \( \{ \tau'_1 \} \) obtained in step (1), or in step (2) from the consideration of previous pairs of states \((s_m, s_n)\) of the \( n \)-th adjacency condition, construct all possible \( k \)-tuples of partial proper partitions containing the pair of states, \((s_m, s_n)\), in one block of \( i \) partitions, \( i=1, \ldots, (k-1) \). Perform the calculations in the following manner:

Let \( \tau'_1 = (B'_1, B'_2) \).

If: \( (s_m, s_n) \cap B'_1 \neq \emptyset \) then construct \( \tau''_1 = (B''_1, B''_2) \) such that \( B''_1 = B'_1 \cup \{ s_m, s_n \} \) and \( B''_2 = B'_2 \).

If: \( (s_m, s_n) \cap B'_2 \neq \emptyset \) then construct \( \tau''_1 = (B''_1, B''_2) \) such that \( B''_2 = B'_2 \cup \{ s_m, s_n \} \) and \( B''_1 = B'_1 \).

If: \( (s_m, s_n) \cap B'_1 = \emptyset \) and \( (s_m, s_n) \cap B'_2 = \emptyset \) then construct \( \tau''_1 = (B''_1, B''_2) \) such that \( B''_1 = B'_1 \cup \{ s_m, s_n \} \), \( B''_2 = B'_2 \) and such \( \tau''_1 = (B''_1, B''_2) \) that \( B''_1 = B_1 \) and \( B''_2 = B_2 \cup \{ s_m, s_n \} \).

Check each constructed \( B_1 \) or \( B_1' \) to see if it satisfies \( |B_1| \leq 2^{k-1} \) or \( |B_1'| \leq 2^{k-1} \) respectively.

(Performing calculations in the way described above, fulfills the constraints 1+3) automatically).

(3) Each partial proper partition \( \tau_1 \) of each family of partial partitions computed in step (1) or step (2), should be checked to see if one of its blocks contains \( 2^{k-1} \) states. In which case, all states \( s \in S \) not contained in any block of \( \tau_1 \) must be connected to the block \( B \) of \( \tau_1 \) except that containing \( 2^{k-1} \) states (constraint 3). Connecting the new states to \( B \) requires that each pair of incompatible states contained in \( B \) to be split into at least one different from \( \tau_1 \) partition from the family considered (constraint 4).

Each computed in step (1), (2) or (3) family of partial proper partitions should be checked to see if it contains only partitions (i.e. for each partition \( \tau_1 \) from this family each state \( s \in S \) is contained in a block of \( \tau_1 \)). If this gives a positive result, then, it means that the FFP has been reached and computations with a
given family must not be continued; otherwise, the computations with a given family must be continued.

Try not to construct the FFP's by combining more than MNAC'k adjacency conditions (constraint 6) and/or by combining more than k adjacency conditions that contain a given state, s_k, as an adjacent state to another state (constraint 5).

These two constraints and the inability to combine a new condition to the ones considered previously are the signals to withdraw from the "branch and bound" process.

In any case, after combining a set of adjacency conditions, a set of k-tuples of partial proper partitions is obtained. If a given k-tuple contains only partial proper partitions which are actually proper partitions, then, such a k-tuple is an FFP. If a given k-tuple contains at least one partial proper partition, τ_i, being not a proper partition, then, it describes a set of k-tuples that contain only proper partitions having been obtained by generating all the proper partitions τ'_i: τ_i ≤ τ'_i for each τ_i (i.e. such τ'_i for which blocks of τ_i are contained in blocks of τ'_i).

Each k-tuple of proper partitions constructed in such a manner is a FFP if it constitutes an orthogonal family of partitions (constraint 4).

**Example 4.8**

Consider the ordered list of adjacency conditions from example 4.7.

Since k=2 and following the rules described above, it is necessary to find a set of pairs of orthogonal partitions according to the adjacency conditions in the order of their ordered list.

Now, consider the first condition: 41

2|3  (1,3)  (1,4)  (2,4)  with the pair of partial proper partitions with empty blocks: { }{ }.

At first, all pairs of partial proper partitions that satisfy condition 2|3 are constructed, i.e. all such pairs that contain states 2 and 3 in different blocks of just one partition.

As a result, one such pair: (2,3)(2,3) is obtained.

Now, check each partial partition from that pair, if this partial proper partition is a partition (i.e. each state s_k is contained in a block of this partition), or if one of the blocks of
this partial proper partition contains $2^{k-1} = 2$ states (in this case all the states not found in any block of the partial partition must be connected to the other block of this partial partition). If the check gives a positive result, then, it means that the FFP has been reached, otherwise, the computations must continue.

In this case, $\{2,3\}(1,4,2,3)$ were obtained. Therefore, computations must continue with $(2,3)(1,4,2,3)$ in order to construct those pairs of partial proper partitions in which the states $(1,3)$ are contained in one block at least of one partition.

Result: $(1,3,2,4)(1,4,2,3)$.

Now, the check gives a positive result - showing that the first FFP has been reached: $\text{FFP}_1 = (1,3,2,4)(1,4,2,3)$.

Further checking of this FFP, displays, that the conditions $(1,4),(2,4); 1\mid 4 : \{1,3,2,4\}, \{1,4,2,3\}, \{1,2,4\} ; 2\mid 4$ and $1\mid 3$ are also satisfied by this FFP.

Consider the next adjacency condition in a similar way, i.e.

\[
1\mid 4 : \{1,4\}(1,4)
\]

after checking the number of states $(1,4)(1,4,2,3)$.

\[
(1\mid 3) : \{1,3,4\}(1,4,2,3)
\]

after checking the number of states: $(1,3,2,4)(1,4,2,3) = \text{FFP}_1$.

Consider the next condition:

\[
3\mid 4 : \{3,4\}(3,4)
\]

after checking the number of states: $(3,4)(1,2,3,4)$.

\[
(1,3) : \{1,3,4\}(1,2,3,4)
\]

after checking the number of states: $(1,3,2,4)(1,2,3,4)$.

The second FFP is reached: $\text{FFP}_2 = (1,3,2,4)(1,2,3,4)$.

Considering all other adjacency conditions, $\text{FFP}_2$ is obtained again.

In this way, the set containing two near optimal FFP's has been constructed:

$\text{FFP}_1 = (1,3,2,4)(1,4,2,3)$ and

$\text{FFP}_2 = (1,3,2,4)(1,2,3,4)$.

The third possible FFP: $\text{FFP}_3 = (1,2,3,4)(1,4,2,3)$ has not been
calculated by this method; so, it would lead to worse assignments than either FFP$_1$ or FFP$_2$.

Check the usefulness of FFP$_1$, FFP$_2$ and FFP$_3$ to the state assignment, by assigning the states using these families of partitions and comparing the results.

$$FFP_1 = (1,3,2,4)(1,4,2,3) :$$

$1,3-\emptyset, 2,4-1, 1,4-\emptyset, 2,3-1$

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>01</td>
<td>11</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>

Assigned next-state table.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Karnaugh map of $S_1$.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Karnaugh map of $S_2$.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Karnaugh map for $Y_1$.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Karnaugh map for $Y_2$. (for $Y_2$ only 2 terms)
If \( \text{FFP}_1 \) is used to assign states, then, the Boolean representation of the next-state and output functions can be implemented with 11 terms - due to common terms.

\[
\text{FFP}_2 = (\overline{1,3},2,4)(\overline{1,2},3,4)
\]

1,3-\( \phi \), 2,4-1, 1,2-\( \phi \), 3,4-1

Assigned next-state table.

\[
\begin{array}{c|cccc}
S & I & 00 & 01 & 11 & 10 \\
---&---&---&---&---&---
1 & 00 & 10 & 11 & 00 & 11 \\
2 & 10 & 11 & 10 & 11 & 10 \\
3 & 01 & 00 & 10 & 00 & 01 \\
4 & 11 & 01 & 10 & 01 & 00 \\
\end{array}
\]

Assigned output table.

\[
\begin{array}{c|cccc}
S & 00 & 01 & 11 & 10 \\
---&---&---&---&---
00 & 1 & 1 & 1 & 0 \\
10 & 0 & 1 & 0 & 0 \\
11 & 0 & 1 & 0 & 0 \\
10 & 1 & 1 & 0 & 1 \\
\end{array}
\]

Karnaugh map of \( S_1 \).

\[
\begin{array}{c|cccc}
S & I & 00 & 01 & 11 & 10 \\
---&---&---&---&---&---
00 & 0 & 0 & 1 & 0 & 1 \\
01 & 0 & 0 & 1 & 0 & 1 \\
11 & 1 & 0 & 0 & 1 & 0 \\
10 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cccc}
S & I & 00 & 01 & 11 & 10 \\
---&---&---&---&---&---
00 & 1 & 1 & 1 & 0 \\
01 & 0 & 0 & 1 & 0 \\
11 & 1 & 1 & 1 & 0 \\
10 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Karnaugh map for \( y_1 \).

\[
\begin{array}{c|cccc}
S & I & 00 & 01 & 11 & 10 \\
---&---&---&---&---&---
00 & 1 & 0 & 0 & 1 \\
01 & 1 & 1 & 0 & 1 \\
11 & 1 & 1 & 0 & 1 \\
10 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Karnaugh map of \( S_2 \).

\[
\begin{array}{c|cccc}
S & I & 00 & 01 & 11 & 10 \\
---&---&---&---&---&---
00 & 0 & 1 & 0 & 1 \\
01 & 0 & 1 & 0 & 1 \\
11 & 1 & 0 & 1 & 0 \\
10 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Karnaugh map for \( y_2 \).
If FFP₂ is used to assign states, then, the Boolean representation of the next-state and output functions can be implemented with 14 terms.

$$\text{FFP}_3 = (\overline{1,4}, \overline{2,3}, 1, 2 - \emptyset, 3, 4 - 1)$$

$$\overline{1,4} - \emptyset, \overline{2,3} - 1, 1, 2 - \emptyset, 3, 4 - 1$$

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

Assigned next-state table.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Karnaugh map of S₁.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Karnaugh map of S₂.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Karnaugh map for Y₁.

Karnaugh map for Y₂.
If FFP\textsubscript{3} is used to assign states, then, the Boolean representations of the next-state and output functions can be implemented with 17 terms.

So, the assignment of states using the first FFP that was found: FFP\textsubscript{1} = (\overline{1}, \overline{3}, 2, 4)\{1, 4, \overline{2}, \overline{3}\} leads to the best result (11 product terms), the second FFP: FFP\textsubscript{2} = (\overline{1}, 3, 2, 4)\{1, 2, 3, 4\}, leads to a worse result with 14 product terms and the FFP not found by this method - FFP\textsubscript{3} leads to the worst result with 17 product terms.

In general, calculations that reach a set of near optimal FFP's based on the ordered list of adjacency conditions are quite simple and implementations of the method presented should be time and memory efficient. However, for extremely large sequential machines, the number of all the adjacency conditions (equal to \(\frac{s(s-1)}{2}\) as well as the number of pairs of states \((s_m, s_n)\) for a given adjacency condition (smaller than or equal to \(|I|\)) can be high and a great number of combinations can be produced. Fortunately, not all of these combinations are practical. Therefore, the whole adjacency conditions as well as the pairs of next-states \((s_m, s_n)\) offering only few adjacencies can be omitted.

When combining adjacency conditions, trade-off must be found between the number of adjacency conditions that can be successfully combined and the number of conditions for next-state pairs \((s_m, s_n)\) which can be satisfied for each adjacency condition. Satisfying more conditions for pairs \((s_m, s_n)\) reduces freedom which is needed for combining more adjacency conditions. One of the possible solutions to this problem is dynamic ordering of adjacency conditions (Chapter 4.9). However, this solution is time-consuming.

Another (static) solution will be obtained when satisfying only the conditions for these pairs \((s_m, s_n)\) will be tried that give many enough adjacencies.

When generating combinations of adjacency conditions, only the conditions with a substantially high number of adjacencies have to be considered as the first conditions in those combinations, i.e. only a number of the first conditions from the ordered list of adjacency conditions.
Finally, just those cases should be considered where pairs of the next-states \((s_m, s_n)\) are contained in one block of a sufficiently high number of partitions \(\tau_1\).

All the simplifications listed above allow considerably fewer combinations of adjacency conditions to be constructed, but these combinations which seem to be the best will still be constructed.

An interesting variation of this method is reached, when considering only those cases when pairs of the next-states \((s_m, s_n)\) are contained in one block of exactly \((k-1)\) partitions \(\tau_1\) for the state assignment. In other words, if a "codewise adjacent" is required from the states \(s_m\) and \(s_n\), i.e. \(s_m \| s_n\).

In this case, a number of adjacencies reached for each pair \((s_m, s_n)\) from each adjacency condition with \(s_k \| s_1\), \(n_{a,m:n}\), will be calculated under the condition that \(s_m\) and \(s_n\) are codewise adjacent, as well as, \(s_k\) and \(s_1\):

\[
n_{a,m:n} = (k-1) \cdot n_{a,m,n} + t_{n_{a,m,n}}
\]

where

\(n_{a,m,n}\) - number of adjacencies reached when the pair of states \((s_m, s_n)\) is contained in one block of one partition \(\tau_1\).

\(t_{n_{a,m,n}}\) - estimated total number of adjacencies reached for the adjacency condition with \(s_m \| s_n\).

Values of the parameters \(n_{a,m,n}\) and \(t_{n_{a,m,n}}\) are given in the ordered list of adjacency conditions.

Then, the ordered list of adjacency conditions has to be changed in the following manner. The pairs of states \((s_m, s_n)\) should be replaced by the pairs of adjacent states \(s_m \| s_n\), the number of adjacencies needed for each pair \((s_m, s_n)\), \(n_{a,m,n}\), should be replaced by the number of adjacencies under the condition \(s_m \| s_n\), \(n_{a,m:n}\), and, for each adjacency condition, the pairs \(s_m \| s_n\) should be ordered from the highest to the lowest values of \(n_{a,m:n}\).
Example 4.9

Consider a variation of the method for the case $s_m | s_n$ and the ordered list of adjacency conditions from Example 4.7.

<table>
<thead>
<tr>
<th>pairs of adjacent states</th>
<th>pairs of adjacent next-states and number of adjacencies</th>
<th>number of unconditionally reached adjacencies</th>
<th>estimation of the total nr. of adjacenc.</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5+22=27</td>
<td>5+14=19</td>
<td>5+13=18</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5+23=28</td>
<td>5+14=19</td>
<td>5+13=18</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7+13=20</td>
<td>5+14=19</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7+14=21</td>
<td>5+13=18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1+16=17</td>
<td>1+15=16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1+16=17</td>
<td>1+15=16</td>
<td></td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions with ordered lists of adjacent next-state pairs.

In this case, the near optimal FFP's are produced in a similar way as before.

In general, adjacency conditions are considered in the order of their ordered list. However, all pairs of adjacent states $s_k | s_1$ already considered, for which not all the adjacency conditions of the next-states were considered, should be indicated and the next pair of adjacent states to be considered should be the pair of next-states to one of the indicated pairs, or the pair of adjacent states from the main list for which the adjacencies are highest.
Example 4.10

In the first place the condition: \(2|3 \ \frac{114}{27}, \frac{114}{19}, \frac{113}{18}\)
will be considered.

\(2|3: (2,3)(2,3)\)
\((2,3)(14,23)\)

\(1|4: (12,34)(14,23)\) or \((13,24)(14,23)\)

Since families of partitions have been reached, the process can stop. However, if continuing then, as the next considered adjacency condition, one of the not remaining pairs of adjacent next-states to \(2|3\) or to \(1|4\) with the highest number of adjacencies should be chosen.

In this case, it is \(\frac{114}{19}\).

Having the value of \(n_{a|m_1n}\) for each pair of states \(s_m|s_n\) from the adjacency condition for \(s_k|s_1\), it is possible to estimate the total number of adjacencies reached for a given condition \(tna(k,1)\) once again.

Firstly, for a given \(s_k|s_1\), all pairs of the next-states \(s_m|s_n\) are ordered according to the value of \(n_{a|m_1n}\) starting from the highest value. Then \(tna(k,1)\) is calculated as follows:

\[
\begin{align*}
&tna(k,1) = \max\left( \\
&\quad \frac{na(k,1)+n_{a|m_1n}}{1+j}, \ \ldots, \ \frac{na(k,1)+\sum_{i=1}^{j} n_{a|m_1n}}{1+j}, \\
&\quad \frac{na(k,1)+\sum_{i=1}^{N_{MNSC}} n_{a|m_1n}}{1+M_{NMSC}} \right)
\end{align*}
\]

Now, it is possible to reorder a list of adjacency conditions using the newly calculated values of \(tna(k,1)\). Such a reordered list may be considered in the same way as that described above, or by further simplification of the method.

The method can be further simplified, if not all the possible combinations of conditions \(s_m|s_n\) that contain up to \(N_{MNSC}\) conditions are considered for a given \(s_k|s_1\), but only those combinations for which the maximum is reached for:
In this way, for a given \( s_k | s_1 \), only the combinations of conditions \( s_m | s_n \), which seem to be the best, will be considered and this will lead to a time and memory efficient algorithm.

**Example 4.11**

Consider the list of adjacency conditions from Example 4.9. Let \( * = \text{na}_a(k,1) \),

\[
\text{na}_a(k,1) + \frac{\sum_{i=1}^{j} \text{na}_m:n_i}{1 + j}, \quad j = \emptyset, 1, \ldots, \text{MNSC}.
\]

Calculation of \( \text{tna}_a(k,1) \) and the best combinations of adjacent next-state pairs.
<table>
<thead>
<tr>
<th>pair of adjacent states</th>
<th>the &quot;best&quot; combinations of pairs of adjacent next-states</th>
<th>estimation of the total number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions.

Now, consider the adjacency conditions in the order given in their ordered list.

2|3 and 1|4:

2|3: \((2,3)(2,3)\)

1|4: \((1,2,3,4)(1,4,2,3) = \text{FFP}_1\) or \((1,3,2,4)(1,4,2,3) = \text{FFP}_2\).

1|4 and 2|3: the same as above.

2|4 and 3|4 and 1|2:

2|4: \((2,4)(2,4)\)

3|4: \((1,2,3,4)(1,3,2,4) = \text{FFP}_3\).

1|3 and 3|4 and 1|2:

1|3: \((1,3)(1,3)\)

3|4: \((1,2,3,4)(1,3,2,4) = \text{FFP}_3\).

3|4 and 1|3 and 2|4:

3|4: \((3,4)(3,4)\)

1|3: \((1,3,2,4)(1,2,3,4) = \text{FFP}_3\).

1|2 and 2|4 and 1|3:

1|2: \((1,2)(1,2)\)

2|4: \((1,3,2,4)(1,2,3,4) = \text{FFP}_3\).

From the example, it is evident that it is possible to simplify the method further. The ordered list of adjacency conditions may be reduced if the adjacency conditions containing the same pairs
of adjacent states is represented by the first such condition on the list.

Example 4.12
In the case considered in Example 4.11, the following reduced list of adjacency conditions is obtained:

<table>
<thead>
<tr>
<th>pairs of adjacent states</th>
<th>estimation of the total number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3 , 1</td>
</tr>
<tr>
<td>2</td>
<td>4 , 3</td>
</tr>
<tr>
<td>1</td>
<td>3 , 3</td>
</tr>
<tr>
<td>3</td>
<td>4 , 1</td>
</tr>
<tr>
<td>1</td>
<td>2 , 2</td>
</tr>
</tbody>
</table>

Reduced ordered list of adjacency conditions.

Further simplification will be obtained, if the adjacency conditions containing only one different pair of adjacent states are represented by the first such condition on the list.

Example 4.13
Reducing the list from Example 4.12, gives the following list.

<table>
<thead>
<tr>
<th>pairs of adjacent states</th>
<th>estimation of the total number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3 , 1</td>
</tr>
<tr>
<td>2</td>
<td>4 , 3</td>
</tr>
</tbody>
</table>

Completely reduced list of adjacency conditions.

Reducing the list of adjacency conditions avoids repeatedly creating the same solutions or (in the case of the second reduction) creating solutions that are estimated to be worse than the previously created ones under the same conditions.

During all the simplifications of the method described above, except the first reduction of the list of adjacency conditions,
some useful information may be lost. Some good solutions may not be reached after these simplifications. On the other hand, for some extremely large machines, the amount of information to be considered in order to generate all the solutions that satisfy the possible combinations of adjacency conditions can be too big for the exact method and implementing the method with the described simplifications will lead to time and memory efficient algorithms. Therefore, not one but a set of implementations of the method should be created: "exact" implementation for medium and large machines and some simplified implementations for extremely large machines.

Finally, the amount of information to be considered depends not only upon the dimensions of a given sequential machine, but also upon their algebraic structure which can be different for different machines of the same size. Therefore, if there are troubles for a given machine with the use of the exact or slightly simplified method; then, the more simplified implementation should be used.
4.7 **The method of maximal adjacencies for non-minimal machines.**

The method of adjacencies can be applied to both minimal and non-minimal machines.

For the latter, the method should be supplemented in the following way.

Firstly, all pairs of compatible states should be found and the respective compatibility conditions must be calculated.

For each pair of compatible states two following cases must be considered:
- a given pair of states will be treated as a pair of compatible states;
- a given pair of states will be treated as a pair of adjacent states.

Thus, the ordered list of adjacency conditions must be supplemented by inserting the pairs of compatible states and the associated compatibility conditions.

The pairs of compatible states must be inserted in the appropriate places on the list, they will result from the benefits of simplifying the implementation of a sequential machine when two given states are combined into one state.

These benefits can be estimated in the following manner.

If two states \( s_k \) and \( s_l \) are combined into one state, one state code combination becomes free. For this state code combination, \( k \cdot |I| \) "don't cares" are obtained in the assigned next-state table and \( |\{Y_i\}| \cdot |I| \) "don't cares" are obtained in the output table. If this state code combination is adjacent to any other combination \( k \cdot |I| + |\{Y_i\}| \cdot |I| \) adjacencies will be reached in the Boolean representations of the next-state and output functions. Additionally, if the states \( s_k \) and \( s_l \) are joined into one state and analyzing input-state dependencies, this pair of states occurs \( ia_{(k,l)} \) times; then, \( k \cdot ia_{(k,l)} \) extra adjacencies will be obtained.

So, combining two compatible states \( s_k \) and \( s_l \) into one state, totally, \( cna_{(k,l)} = k \cdot |I| + |\{Y_i\}| \cdot |I| + k \cdot ia_{(k,l)} \) adjacencies will be obtained.

If the two states \( s_k \) and \( s_l \) are combined into one state, then, the adjacencies of all the states \( s_p: s_p \neq s_k \text{ and } s_p \neq s_l \), with the new state \( (s_k \sim s_l) \), should be considered.
Thus, for each pair of compatible states \((s_k - s_1)\) and all states \(s_p: s_p \neq s_k\) and \(s_p \neq s_1\), the adjacency conditions must be created and ordered. This can be done in exactly the same way as for the state adjacency conditions considered before, but the state-table of a sequential machine should be replaced before with the state-table of a partially minimized machine where the states \(s_k\) and \(s_1\) are connected together. In a result for every pair of compatible states \((s_k - s_1)\), an ordered list of adjacency conditions with this pair will be obtained. The adjacencies described by this list will be reached under the condition that two given states \(s_k, s_1\) are combined into one state (a given compatibility \(s_k - s_1\) is active) and a given adjacency condition for \((s_k - s_1) | s_p\) is satisfied.

When placing a given pair of compatible states \((s_k - s_1)\) on the ordered list of adjacency conditions, not only the number of adjacencies, \(c_{na(k,1)}\) that are reached for these states under the condition that they are combined into one state, have to be taken into account, but also the number of adjacencies that are obtained for the adjacency condition with the highest number of adjacencies from the list of adjacency conditions with the pair \((s_k - s_1) - \max(t_{na(k-1),p})\).

The pairs of compatible states should be put on the ordered list of adjacency conditions at the appropriate places as indicated by the value of the \(t_{na(k-1)}\) parameter.

Near optimal FFP's are generated based on the ordered list of adjacency conditions with pairs of compatible states and on the ordered lists of adjacency conditions with given pairs of compatible states.

When combining the adjacency conditions in order to form FFP's for non-minimal machines, we must remember that only the first two constraints formulated for minimal machines remain valid. The other four constraints are changed in the following manner:

(3) Not only the proper partitions but all two-block partitions, can be useful for the state assignment of a non-minimal
(4) The condition of orthogonality is replaced with the separation condition: each two incompatible states of the machine must be separated by at least one partition from a FFP.

(5) Each state \( s_k \in S \) may be codewise adjacent with, at most, \( k + |\text{sapcs}| \) other states, if \( s_k \not\in \text{sapcs} \), or \( k + |\text{sapcs}| - 1 \) other states if \( s_k \in \text{sapcs} \), where \( \text{sapcs} \) – the set of active pairs of compatible states.

(6) The number of adjacency conditions which can be fulfilled together for a given number of states \(|S|\), a given number of active pairs of compatible states \(|\text{sapcs}|\) and a given assignment length \(k\), is not greater than \(MNAC \leq M_{NAC}^{(S|-\text{sapcs}|)}\) (adjacency conditions with the states in the active pairs of compatible states being taken into account only once – as adjacency conditions with given pairs of compatible states).

A limited set of near optimal FFP's is constructed for non-minimal machines in nearly the same way as for minimal machines but, additionally, pairs of compatible states must be considered. The differences are described by the constraints (3)+(6) given above and by the additional constraint (7):

(7) The closure condition must be satisfied, i.e. for each pair of conditionally compatible states, all compatibility conditions must be fulfilled.

If two conditionally compatible states are considered; then, all the compatibility conditions must be satisfied before the other pairs of compatible states or adjacent states can be considered.

If, for a given constructed family of partial two-block partitions \( \{\tau_i\} \) and a given pair of states \( (s_k, s_l) \) being candidates to be compatible, satisfaction of at least one compatibility condition is impossible; then, the states \( s_k \) and \( s_l \) cannot be combined as a pair of compatible states with a constructed family of partial partitions \( \{\tau_j\} \), i.e. the adjacency condition that contains the pair \( (s_k, s_l) \) cannot be satisfied for \( \{\tau_i\} \).

If, among successfully combined conditions, the conditions that contain some pairs of compatible states are present; then,
such pairs of compatible states are said to be active and the lists of adjacency conditions for such pairs of states are said to be active.

Both the adjacency conditions from the main list of adjacency conditions and the conditions from the active lists of adjacency conditions, with pairs of compatible states, should be taken into account when constructing the near optimal FFP's.

As the next condition to be considered, the condition from one of the active lists, or from the main list, offering the highest number of adjacencies, should be chosen.

The method of maximal adjacencies applied to non-minimal sequential machines generates not only a limited set of near optimal state assignments, but also it reduces the number of machine states by assigning the same codes to some of the compatible states.

Application of this method to non-minimal sequential machines is illustrated with the following example.
Example 4.14
Let $k=2$.

<table>
<thead>
<tr>
<th>$S$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_3$</th>
<th>$x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>

Next-state table.

| $S$ | $x_0|x_j$ | $x_1|x_3$ | $x_3|x_2$ | $x_2|x_0$ |
|-----|----------|----------|----------|----------|
| 0   | -        | -        | -        | -        |
| 1   | 1        | 1,3      | 1,3      | -        |
| 2   | 2,3      | -        | -        | 1,3      |
| 3   | 0,1      | 0,1      | 1,4      | 1,4      |
| 4   | -        | -        | -        | -        |

Table of adjacency conditions for input-state dependencies.

<table>
<thead>
<tr>
<th>block of states</th>
<th>number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>2</td>
</tr>
<tr>
<td>1,3</td>
<td>3</td>
</tr>
<tr>
<td>1,4</td>
<td>2</td>
</tr>
<tr>
<td>2,3</td>
<td>1</td>
</tr>
</tbody>
</table>

List of adjacency conditions for input-state dependencies.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and number of their occurrences</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1,2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

List of adjacency conditions from state--next-state dependencies.
### List of Adjacency Conditions for State-Output Dependencies

<table>
<thead>
<tr>
<th>Adjacent States</th>
<th>Next-State Blocks and Number of Their Occurrences</th>
<th>Number of Independently Reached Adjacencies</th>
<th>Number of &quot;Don’t Cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0,3</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0,3</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0,1</td>
<td>1,4</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6 + 1 = 7</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0,1</td>
<td>2 + 2 = 4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1,3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3</td>
<td>0,2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0,3</td>
<td>1</td>
</tr>
</tbody>
</table>

Converted list of adjacency conditions from state--next-state dependencies.
<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and number of their occurrences</th>
<th>number of independently reached adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0,3 3,4</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0,3</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0,1 1,4</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0,1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1,3</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3 0,2 1,4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,3 2,3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0,3</td>
<td>11</td>
</tr>
</tbody>
</table>

Common list of adjacency conditions.

<table>
<thead>
<tr>
<th>pair of adjacent states</th>
<th>estimation of the total number of adjacencies</th>
<th>pair of adjacent states</th>
<th>estimation of the total number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>12 + 1 = 13</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>13 + 1 = 14</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>10 + 3 = 13</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>15</td>
<td>3</td>
</tr>
</tbody>
</table>

Estimation of the total number of adjacencies for MNSC = k-1 = 1 and MNP = [k/2] = 1.
### Adjacent States and Number of Their Occurrences

<table>
<thead>
<tr>
<th>Adjacent States</th>
<th>Blocks of Next-States and Number of Their Occurrences</th>
<th>Number of Unconditionally Reached Adjacencies</th>
<th>Estimation of the Total Number of Adjacencies</th>
<th>Number of &quot;Don't Cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1,3</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0,3</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0,3 3,4</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0,1 1,4</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0,1</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0,3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,3 2,3</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3 1,4 0,2</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions.

From the output and next-state tables, it follows that the states 0,4 and 1,2 are unconditionally compatible.

So, the compatibility conditions for 0-4 and 1-2 must be constructed.

<table>
<thead>
<tr>
<th>Pair of Compatible States</th>
<th>Number of Adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 4</td>
<td>8 + 8 = 16</td>
</tr>
<tr>
<td>1 - 2</td>
<td>8 + 8 = 16</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
</tr>
<tr>
<td>(0-4)</td>
<td>(0-4)</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Partially minimized next-state table for 0-4.

<table>
<thead>
<tr>
<th>I</th>
<th>S</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
<td>(1-2)</td>
<td>3</td>
<td>(1-2)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(1-2)</td>
<td>0</td>
<td>(1-2)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Partially minimized next-state table for 1-2.

<table>
<thead>
<tr>
<th>adjacency condition</th>
<th>number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4),1</td>
<td>4</td>
</tr>
<tr>
<td>(0-4),3</td>
<td>4</td>
</tr>
</tbody>
</table>

Adjacency conditions with 0-4 from input-state dependencies.

<table>
<thead>
<tr>
<th>adjacency condition</th>
<th>number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2),0</td>
<td>2</td>
</tr>
<tr>
<td>(1-2),3</td>
<td>4</td>
</tr>
<tr>
<td>(1-2),4</td>
<td>2</td>
</tr>
</tbody>
</table>

Adjacency conditions with 1-2 from input-state dependencies.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and nr. of their occurrences</th>
<th>nr. of &quot;dc&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>1</td>
<td>(0-4),3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>(0-4)</td>
<td>2</td>
<td>(0-4),3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>(0-4)</td>
<td>3</td>
<td>(0-4),1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Adjacency conditions with 0-4 from state--next-state dependencies.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and nr. of their occurrences</th>
<th>nr. of &quot;dc&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>0</td>
<td>0,3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
<td>(1-2),3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>(1-2)</td>
<td>4</td>
<td>(1-2),3</td>
</tr>
</tbody>
</table>

Adjacency conditions with 1-2 from state--next-state dependencies.

N.B. * = (1-2),0 ; ** = (1-2),4

Adjacency conditions with 1-2 from state--next-state dependencies.
Reduced output table for $\emptyset$-4. Reduced output table for 1-2.

<table>
<thead>
<tr>
<th>$S$</th>
<th>0</th>
<th>1</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>01</td>
<td>—</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>0</th>
<th>1</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>10</td>
<td>—</td>
<td>10</td>
</tr>
</tbody>
</table>

Adjacency conditions with $\emptyset$-4 Adjacency conditions with 1-2 from state-output dependencies. from state-output dependencies.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>1</td>
</tr>
<tr>
<td>(0-4)</td>
<td>2</td>
</tr>
<tr>
<td>(0-4)</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adjacency states</th>
<th>number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>0</td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
</tr>
<tr>
<td>(1-2)</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and number of their occurrences</th>
<th>number of independently reached adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>(0-4), 3</td>
<td>1, 3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0-4)</td>
<td>(0-4), 3</td>
<td>2, 3</td>
<td>1, 3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(0-4)</td>
<td>(0-4), 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Common list of adjacency conditions with $\emptyset$-4.
<table>
<thead>
<tr>
<th>adjacent states</th>
<th>next-state blocks and number of their occurrences</th>
<th>number of independently reached adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>0</td>
<td>0,3</td>
<td>3,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
<td>(1-2),0</td>
<td>(1-2),4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(1-2)</td>
<td>4</td>
<td>(1-2),3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Common list of adjacency conditions with 1-2.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>estimation of the number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>1</td>
</tr>
<tr>
<td>(0-4)</td>
<td>2</td>
</tr>
<tr>
<td>(0-4)</td>
<td>3</td>
</tr>
</tbody>
</table>

Estimation of the total number of adjacencies with 0-4.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>estimation of the number of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>0</td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
</tr>
<tr>
<td>(1-2)</td>
<td>4</td>
</tr>
</tbody>
</table>

Estimation of the total number of adjacencies with 1-2.

<table>
<thead>
<tr>
<th>adjacent states</th>
<th>blocks of next-states and number of their occurrences</th>
<th>number of independently reached adjacencies</th>
<th>estimation of the total number of adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0-4)</td>
<td>3</td>
<td>(0-4),1</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0-4)</td>
<td>1</td>
<td>(0-4),3,1,3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0-4)</td>
<td>2</td>
<td>(0-4),3,2,3,1,3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions with 0-4.
| adjacent states | blocks of next-states and number of their occurrences | number of independently reached adjacencies | estimation of the total number of adjacencies | number of "don't cares"
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1-2)</td>
<td>4</td>
<td>(1-2), 3</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1-2)</td>
<td>3</td>
<td>(1-2), 0</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1-2)</td>
<td>0</td>
<td>0, 3</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions with 1-2.
<table>
<thead>
<tr>
<th>adjacent states</th>
<th>blocks of next-states and number of their occurrences</th>
<th>number of unconditionally reached adjacencies</th>
<th>estimation of the total number of adjacencies</th>
<th>number of &quot;don't cares&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1,3</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0,3</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0,3</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0,1</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0,1</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0,3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1,3</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1,3</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Ordered list of adjacency conditions with inserted pairs of compatible states.
Let us construct now a set of FFP's based on the ordered list of adjacency and compatibility conditions.
We start from the first condition:

\[
\begin{array}{l}
1|4: (1,3) \\
1|4: (1,4)(1,4) \\
(1,3): (1,3,4)(1,4,3) \quad (1,3 \text{ are not compatible}) \\
0-4: (1,3,0,4)(0,1,4,3) \\
1-2: (1,2,3,0,4)(0,1,2,4,3) = \text{FFP}_1 \\
(1,2,3,0,4) \cdot (0,1,2,4,3) = (0,4,1,2,3) \\
\text{completely minimized 3-state implementation} \\
(0-4)|3 \quad \text{(from the ordered list of adjacency conditions with 0-4)}: \\
\text{impossible (0-4) and 3 are splitted in both partial partitions).} \\
1|2: \text{impossible (separation condition can not be satisfied for 2 and 3 or for 2 and 4).} \\
0|2: \text{impossible} \\
0|1: \text{satisfied} \\
(0,3) \text{ and (3,4): impossible} \\
1|3: \text{satisfied} \\
(0,1): \text{satisfied} \\
3|4:\text{impossible} \\
2|4: (1,3,0,2,4)(0,1,4,2,3) = \text{FFP}_2 \\
(1,3,0,2,4) \cdot (0,1,4,2,3) = (0,4,1,2,3) \\
\text{partially minimized 4-state implementation} \\
2|3 (1|3) (1|4): \text{also satisfied for FFP}_2. \\
1-2: (1,2,3,4)(1,2,4,3) \\
0|4: (1,2,3,0,4)(1,2,4,0,3) = \text{FFP}_2 \\
(1,2,3,0,4) \cdot (1,2,4,0,3) = (0,1,2,3,4) \\
\text{partially minimized 4-state implementation} \\
\end{array}
\]

Let us start now from condition 0-4.

\[
\begin{array}{l}
0-4: (0,4)(0,4) \\
1-2: (0,4,1,2)(0,4,1,2) \quad \text{,} \quad (0,1,2,4)(0,4,1,2) \\
\end{array}
\]
Let us start now from condition 1-2:

\[ 0|2 \begin{array}{c} \frac{0,3}{1} \\ \hline \end{array} \]

\[
\begin{align*}
0|2: & \quad (0,1,2,4)(0,4,1,2) \\
(0,3): & \quad (0,1,2,4,3)(0,3,4,1,2) = \text{FPP}_4 \\
\text{completely minimized 3-state implementation} \\
0\overline{4}: & \quad (0,4)(0,4) \\
1\overline{2}: & \quad (0,1,4,2)(0,4,1,2), (0,2,4,1)(0,4,1,2) \\
(0\overline{4})|3 \begin{array}{c} \frac{0\overline{4}}{6} \\ \hline \end{array} \quad \text{(from the ordered list of adjacency conditions with 0\overline{4})}:
\]

\[
\begin{align*}
(0\overline{4})|3: & \quad (0,1,4,2,3)(0,3,4,1,2), (0,2,4,1,3)(0,3,4,1,2) \\
((0\overline{4}),1): & \quad (0,1,4,2,3)(0,3,4,1,2) = \text{FPP}_5 \\
\text{partially minimized 4-state implementation} \\
0\overline{4}: & \quad (0,4)(0,4) \\
1\overline{2}: & \quad (0,1,4,2)(0,4,1,2), (0,2,4,1)(0,4,1,2) \\
(0\overline{4})|3 \begin{array}{c} \frac{0\overline{4}}{6} \\ \hline \end{array} \quad \text{(from the ordered list of adjacency conditions with 0\overline{4})}:
\]

\[
\begin{align*}
(0\overline{4})|3: & \quad (0,1,4,2,3)(0,3,4,1,2), (0,2,4,1,3)(0,3,4,1,2) \\
((0\overline{4}),1): & \quad (0,1,4,2,3)(0,3,4,1,2) = \text{FPP}_5 \\
\text{partially minimized 4-state implementation} \\
0\overline{4}: & \quad (0,4)(0,4) \\
1\overline{2}: & \quad (0,1,4,2)(0,4,1,2), (0,2,4,1)(0,4,1,2) \\
(0\overline{4})|3 \begin{array}{c} \frac{0\overline{4}}{6} \\ \hline \end{array} \quad \text{(from the ordered list of adjacency conditions with 0\overline{4})}:
\]

Let us start now from condition 1-2:

\[ 1\overline{2}: \quad (1,2), (1,2) \\
0\overline{4}: \quad (0,1,2,4)(1,2,0,4), (1,2,4,0)(1,2,0,4) \\
0|2 \begin{array}{c} \frac{0,3}{1} \\ \hline \end{array} \quad \text{(or } 1\overline{2})|4 \begin{array}{c} \frac{1\overline{2}}{6} \\ \hline \end{array} \)

\[
\begin{align*}
0|2: & \quad (0,1,2,4)(1,2,0,4) \\
(0,3): & \quad (0,1,2,3,4)(1,2,3,0,4) = \text{FPP}_6 \\
\text{partially minimized 4-state implementation} \\
0\overline{4}: & \quad (0,4)(0,4) \\
1\overline{2}: & \quad (0,1,4,2)(0,4,1,2), (0,2,4,1)(0,4,1,2) \\
(0\overline{4})|3 \begin{array}{c} \frac{0\overline{4}}{6} \\ \hline \end{array} \quad \text{(from the ordered list of adjacency conditions with 0\overline{4})}:
\]

This process can be continued in order to consider all the combinations of conditions that contain at least one compatibility condition (since \(k=2\), the number of states should be reduced to no more than 4, i.e. at least one compatibility condition should be satisfied by a FPP). However, the first five reached FPP's are the best.

Now, assign the next-state and output tables with obtained FPP's and compare the results.
\[
\begin{align*}
FPP_1 &= (1,2,3,0,4)(0,1,2,4,3) \\
FPP_2 &= (1,3,0,2,4)(0,1,4,2,3) \\
FPP_3 &= (1,2,3,0,4)(1,2,4,0,3) \\
FPP_4 &= (0,1,2,4,3)(0,3,4,1,2) \\
FPP_5 &= (0,1,4,2,3)(0,3,4,1,2) \\
FPP_6 &= (0,1,2,3,4)(1,2,3,0,4)
\end{align*}
\]

<table>
<thead>
<tr>
<th>STATE</th>
<th>FPP_1</th>
<th>FPP_2</th>
<th>FPP_3</th>
<th>FPP_4</th>
<th>FPP_5</th>
<th>FPP_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
<td>(S_1S_2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Because of common terms for FFP₁, a 12-term implementation is obtained.
### FFP$_2$:

<table>
<thead>
<tr>
<th>$S_1S_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 - 00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>2 - 11</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>3 - 01</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>4 - 10</td>
<td>01</td>
<td>01</td>
<td></td>
<td>01</td>
</tr>
</tbody>
</table>

**Assigned next-state table.**

<table>
<thead>
<tr>
<th>$S_1S_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

**Reduced next-state table.**

<table>
<thead>
<tr>
<th>$S_1S_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Karnaugh map for $S_1$.**

<table>
<thead>
<tr>
<th>$S_1S_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Karnaugh map for $S_2$.**

<table>
<thead>
<tr>
<th>$S_1S_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Karnaugh map for $Y_1$.**

**Karnaugh map for $Y_2$.**

**For FFP$_2$ a 13-term implementation is obtained.**
### FFP₃:

<table>
<thead>
<tr>
<th>S₁S₂</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>01</td>
<td></td>
<td>01</td>
</tr>
</tbody>
</table>

### Assigned next-state table:

<table>
<thead>
<tr>
<th>S₁S₂</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td></td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

### Reduced next-state table:

<table>
<thead>
<tr>
<th>S₁S₂</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Karnaugh map for S₁:

Because of common terms for FFP₃, an 11-term implementation is obtained.
Because of common terms in the Boolean representations in the next-state and output functions for FFP₄, a 9-term implementation is obtained.
Because of common terms for $\text{FFP}_5$, an 11-term implementation is obtained.
Because of common terms for FFP₆, an 11-term implementation is obtained.
The following assignment results were obtained:

**FFP_4** (completely minimized 3-state machine): 9-term implem.  
**FFP_3** (partially minimized 4-state machine): 11-term implem.  
**FFP_5** (partially minimized 4-state machine): 11-term implem.  
**FFP_6** (partially minimized 4-state machine): 11-term implem.  
**FFP_1** (completely minimized 3-state machine): 12-term implem.  
**FFP_2** (partially minimized 4-state machine): 13-term implem.

All the FFP's that were constructed gave good assignment results. The best result was reached for one of the assignments minimizing a given machine (FFP_4) completely; however, very good results were reached for three other assignments too, namely, minimizing the machine partially (FFP_3, FFP_5, FFP_6) — these results were better than the result obtained for another assignment that minimizes the machine (FFP_1) completely.
4.8 Conditions for the common terms.

The combinational network implementing the next-state and output functions of a sequential machine is a multiple-output circuit.

Therefore, not only the small number of product terms for each of the Boolean functions representing the next-state and output functions is important for obtaining the low complexity realization, but also the high number of common product terms for two or more of these Boolean functions.

Although the method of maximal adjacencies is used only in order to pre-select assignments, constructing a limited size set of near optimal assignments, which is then checked considering different types of flip-flops and using a multiple output logic minimizer, it is possible to take into account the common terms not only during multiple output minimization but also in the method of maximal adjacencies itself.

Common terms are obtained when for codewise adjacent states and a given input or for codewise adjacent inputs and a given state adjacent "1"s or "0"s appear simultaneously in two or more Boolean functions representing next-state and output functions.

4.8.1 Common terms for Boolean output functions.

Since only sequential machines are considered here, for which the inputs and outputs are assigned, the dependency of output variables of input variables and of states is predefined and the codewise adjacency of states is the only parameter which can be changed in order to obtain more common terms in the functions describing output variables.

It is possible to give a priority for those pairs of adjacent states for which adjacent "1"s or "0"s are obtained in more than one Boolean output function.

Such pairs are preferred in a natural way by the method; if adjacencies are in more functions, then it is simply more of them and the method, maximizing the total number of adjacencies, prefers such cases. However, if from the practical use of the method it follows that such pairs of adjacent states should be more preferred, then there is no objection to do that (for example by giving higher weights for common adjacencies).
4.8.2 Common terms for Boolean next-state functions.

For a given pair of adjacent states $s_k|s_1$ and a given input $x_j$, adjacent "0"s or "1"s are obtained simultaneously in more than one Boolean next-state function in the following cases:

- "don't care" is obtained in the table of adjacency conditions from the present-state--next-state dependencies, i.e.
  \[ \delta(s_k, x_j) = \delta(s_1, x_j) \text{ or } \delta(s_k, x_j) = "-" \text{ or } \delta(s_1, x_j) = "-" \text{ or } \delta(s_k, x_j) = "-" \text{ and } \delta(s_1, x_j) = "-" \] (adjacencies are obtained simultaneously in $k$ Boolean next-state functions);

- a pair of next-states $(s_m, s_n)$, where $s_m = \delta(s_k, x_j)$ and $s_n = \delta(s_1, x_j)$ is contained in one block of more than one partition used for state assignment (adjacencies are obtained in as many Boolean next-state functions as many partitions contain a given pair $(s_m, s_n)$ in one block).

For a given pair of adjacent inputs $x_i|x_j$ and a given state $s_k$, adjacencies are obtained simultaneously in more than one Boolean next-state function if a pair of next-states $(s_m, s_n)$, where $s_m = \delta(s_k, x_i)$ and $s_n = \delta(s_k, x_j)$, is contained in one block of more than one partition used for state assignment (adjacencies are obtained in as many Boolean next-state functions as many partitions contain a given pair $(s_m, s_n)$ in one block).

The following conclusion follows from the observations above:
1) the pairs of adjacent states $s_k|s_1$ with many "unconditional" adjacencies should be preferred;
2) the appearance of pairs of states $(s_m, s_n)$ in one block of many partitions used to state assignment should be preferred.

As for Boolean output functions, the pairs of adjacent states $s_k|s_1$ with many "unconditional" adjacencies are preferred by the method in a natural way and if they should be more preferred then there is no objection to that (for example by giving higher weights to these adjacencies).

Appearance of state pairs $(s_m, s_n)$ in one block of many partitions is also preferred automatically by the method, especially by its simplified version where states $s_m$ and $s_n$ are considered as codewise adjacent, i.e. the pair $(s_m, s_n)$ must
appear in one block of (k-1) of k partitions used for state assignment (adjacencies are reached simultaneously in (k-1) Boolean next-state functions).

It should be stressed, however, that the possibility of common terms existing in several Boolean next-state functions is limited, because the task of these functions is to distinguish states using a possibly small number of memory elements and not to describe their common features. Many practical examples confirm this intuition. In most cases, the Boolean next-state functions possess only a few common terms.

4.8.3 Common terms for Boolean next-state and output functions.

The problem of obtaining common terms for Boolean next-state and output functions is important, because it is possible to reach many such common terms in most practical cases and to simplify considerably the realization of a sequential machine. For many Moore machines, especially for counters and counter-like machines, it is possible to assign a state variable \( S_i \) according to the partition \( r_i \) forming with a given output partition \( \omega_k \) a S-O partition pair, where \( \omega_k \) is a partition induced by values "0" and "1" of an output variable \( Y_k \) on the set of outputs \( O \).

In such a case, for all internal states \( s: s \in S \), values of the output variable \( Y_k \), that introduces \( \omega_k \), will be the same as values of the state variable \( S_i \) assigned according to \( r_i \) or the same as negated values of \( S_i \). So, the Boolean function \( \lambda_k \) for describing values of \( Y_k \) is the same as the Boolean function \( \delta_i \) for describing values of \( S_i \), or it is the negation of \( \delta_i \), i.e. all terms of those two functions are common. In this case, the implementation of the Boolean output function \( \lambda_k \) is trivially simple and cost-free.

In the case of a Mealy machine, such a big simplification of the output function \( \lambda \), at least for some of the Boolean functions \( \lambda_k \), is not possible, because, in a Mealy machine, outputs depend not only of states but of inputs as well. However in this case, very often, many common terms for Boolean next-state and output functions can also be reached, greatly simplifying the Boolean representation of a machine.

In order to consider the possibility of common terms in Boolean next-state and output functions of a Mealy machine, the method of
maximal adjacencies should be implemented, taking into account the two described below factors having direct influence on the number of common terms.

The first parameter for given adjacent states \( s_k \mid s_1 \) and all \( x_i \in I \), and all Boolean output functions \( \lambda_k \), is the number of the next-state pairs \( (\delta(s_k, x_i), \delta(s_1, x_i)) \) that satisfy the following conditions:

\[
\delta(s_k, x_i) = \delta(s_1, x_i) \quad \text{or} \quad \delta(s_k, s_1) = "-" \quad \text{or} \quad \delta(s_1, x_i) = "-"
\]

and

\[
\lambda_k(s_k, x_i) = \lambda_k(s_1, x_i) \quad \text{or} \quad \lambda_k(s_k, x_1) = "-" \quad \text{or} \quad \lambda_k(s_1, x_i) = "-".
\]

If the conditions above are satisfied for two adjacent states \( s_k \mid s_1 \) and a given input \( x_i \), then, coding the blocks of partitions \( \tau_i \) containing states \( \delta(s_k, x_i) \) and \( \delta(s_1, x_i) \) adequately with "1" or "0", adjacent ones or zeroes will be obtained simultaneously in the Boolean next-state functions \( \{\delta_i\} \) describing the next-state variables \( \{S_i\} \) that are assigned according to \( \{\tau_i\} \) and in the Boolean output function \( \lambda_k \).

The pairs \( s_k \mid s_1 \) with high values of this parameter are automatically preferred by the method because, if the conditions above are satisfied, there will be more "unconditional" adjacencies for a given pair of adjacent states \( s_k \mid s_1 \). However, from the practical use of the method, it can follow that such pairs of adjacent states should be more preferred. This can be done by adding the value of the parameter with the appropriate weight to the number of "independent" adjacencies for \( s_k \mid s_1 \).

The second parameter for a given pair of states \( (s_m, s_n) \) is the number of "1"s or "0"s that are reached in a given Boolean next-state function \( \delta_i \) coincidentally with the Boolean output functions and the adequate number of lost common "0"s or "1"s if the block of a partition \( \tau_i \) containing a pair of states \( (s_m, s_n) \) is coded adequately with "1" or "0".

This parameter can be calculated as follows:

1. For each (next) state \( s_m \): \( s_m \in S \), the number of times this state occurs in the next-state table in coincidence with the value "1" of an output variable \( y_k \) and the number of occurrences in coincidence with the value "0" of \( y_k \) is calculated for each output variable \( y_k \).
2. For each pair of (next)states \((s_m, s_n)\), the number of times this pair coincides with the "1"s of a given output function \(\lambda_k\) and the number of times this pair coincides with the "0"s of \(\lambda_k\) should be calculated for each \(\lambda_k\), adding the calculated in point (1) values for \(s_m\) and \(s_n\).

3. For each pair of states \((s_m, s_n)\), the difference between the number of "1"s reached in coincidence with a given output function \(\lambda_k\) and the number of "lost" coincident "0"s under the condition, that a given pair of states \((s_m, s_n)\) that is contained in the block of a partition \(\tau_i\) coded with "1", should be calculated for each \(\lambda_k\) (as a difference of two parameters calculated at point (2)).

A positive value for the parameter at point (3) says that by coding a given pair \((s_m, s_n)\) with "1" as more common "1"s for the next-state function \(\delta_i\) and for the output functions \(\lambda_k\) are obtained than common "0"s by coding this pair with "0" as the value of the parameter is.

A negative value means that when coding a given pair \((s_m, s_n)\) with "0" then the modulus of the parameter common "0"s more than common "1"s are obtained. So, the pairs of states \((s_m, s_n)\), for which the sum of positive values of this parameter for all \(\lambda_k\) is high, should be contained in blocks of partitions that are coded with "1" and the pair \((s_m, s_n)\), for which the sum of negative values is high, contained in blocks of partitions coded with "0". If both the sum of positive and of negative values is great, then, in some of the partitions used for assignments of a given pair \((s_m, s_n)\) will be contained in blocks coded with "1" and in blocks coded with "0" for the rest of the partitions. If both sums are small, then from the point of view of common "1"s or "0"s, it is unnecessary to keep the states \(s_m\) and \(s_n\) in the same blocks of partitions that are used for the state assignment.

4. The sum of the modules of values that are calculated for a given \((s_m, s_n)\) in point (3) should be calculated for all \(\lambda_k\).

Utilizing the parameter calculated in point 4 is very simple. Its value should be simply added, with appropriate weight, to the number of adjacencies for pairs \((s_m, s_n)\), pairs
$s_k s_1$ (multiplying by $(k-1)$) and $s_k - s_1$ (multiplying by $k$) in the common list of adjacency conditions.

The sign of a greater (positive or negative) value of sums for a given $(s_m, s_n)$ and all $1_k$, can also be utilized. It can be used not only as an indication of how to code blocks of partitions containing a given pair of states $(s_m, s_n)$, but also of how to construct these partitions (pairs of states with the same sign should be in the same block of a partition that is used for the state assignment).

**Example 4.15**

Let us consider the problem of common terms in the Boolean next-state and output functions of the sequential machine from Example 4.14.

<table>
<thead>
<tr>
<th>$s_k s_1$</th>
<th>Number of next-state adjacencies coincidental with output-adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0-4</td>
<td>8</td>
</tr>
<tr>
<td>1-2</td>
<td>8</td>
</tr>
</tbody>
</table>

From the viewpoint of common terms in the Boolean next-state and output functions, the compatibility of states 0-4 and 1-2 and the adjacency of states 0|4, 0|2, 1|2 and 1|4 should have a priority.

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>Number of coincidental occurrences with output value 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$y_2$</th>
<th>Number of coincidental occurrences with output value 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
From the table above, it follows that pairs of states 0,3, 1,2, 3,4, 0,1, 2,3 and 1,4 should have a priority, from the viewpoint of common terms in the Boolean next-state and output functions.

It is evident now, why so many common terms were reached in Example 4.14 for:

\[ \text{FFP}_4 = (0,1,2,4,3)(0,3,4,1,2) \] and
\[ \text{FFP}_5 = (0,1,4,2,3)(0,3,4,1,2). \]

If the information above about the possibility to generate common terms is used to generate FFP's then FFP_4 and FFP_5 will be generated as the first two FFP's.
4.9 Dynamic ordering of adjacency conditions.

The static estimation of the number of adjacencies that are obtained when a given adjacency condition is satisfied, has been considered before. In this paragraph, a dynamic estimation of this parameter and the dynamic ordering of adjacency conditions will be considered.

Following the description given before, the near optimal final families of partitions are created by combining adjacency conditions in the order defined by their statically ordered list. However, the choice of the next condition may not be so strictly predefined. It may depend on the choices made before.

Let us consider now the simplified version of the method of adjacencies, where the adjacency of states from the next-state pairs is required (the considerations for a general version of the method are similar).

Each component condition of a given adjacency condition having the form $s_i \mid s_j$, i.e. each condition that two states $s_i \mid s_j$ must be adjacent, will be called the primitive adjacency condition.

A given primitive adjacency condition is active if it is satisfied by a given family of partial two-block partitions being constructed, i.e. if it has been used in order to create this family of partial partitions.

If a given primitive adjacency condition $s_i \mid s_j$ is active then we know that $s_k$ and $s_1$ are adjacent and we can use this fact in order to obtain a better estimation of the number of adjacencies obtained for each of the adjacency conditions containing a given primitive condition. For each adjacency condition containing a given primitive adjacency condition, we must simply calculate a new value for $t_{na}$, taking into account that the primitive condition is satisfied. Then, the list of adjacency conditions has to be reordered according to the results of the performed calculations.

Moreover, such a primitive adjacency condition (from the main list of adjacency conditions containing present-state pairs or from one of the lists of conditions containing the next-state pairs for the active adjacency conditions) has to be chosen as the next active primitive condition, that maximizes the current number of adjacencies, taking into account all the currently active primitive adjacency conditions.
5. Conclusions.

The problem of, in the strict sense, minimal realization of sequential machines is entirely unsolved (except complete enumeration), and there is no indication at present that a solution might be found in the near future.

Typically, this problem can be replaced with a number of subproblems such as: state minimization, state assignment, choice of flip-flops and minimization of Boolean functions representing the next-state and output functions of a sequential machine.

In this work, the greatest attention has been paid to state assignment and state minimization; however, the method, as a whole, covers all the subjects listed.

Two traditionally independent steps: state minimization and state assignment are replaced here with a single process of concurrent state minimization and assignment. Here, minimization (or partial minimization) of internal states is obtained as a byproduct of the state assignment and it results from assigning the same code to two or more internal states.

The problem of, in the strict sense, optimal state assignment is unsolved, but some approximate approaches have been proposed. The best known of them are: the partition theory [7][8][12][13][16][22][24][25], the column based approach [4], the graph embedding approach [1][2][15] and related to it multi-valued, multi-output, non-univocal function minimization methods [3][20][21].

Algebraic methods for state assignment based on partition theory and the reduced dependency theorem, developed by Hartmanis [7][8][12][13], Stearns [8][12][13], Karp [16], Potossin [24] and others, suffer from the following shortcomings:

- they try to reduce the number of arguments (state and input variables) for Boolean functions, that describe the combinational part of a sequential machine (this criterion is often useless because realizations with a minimal number of variables can have more product terms and more active product terms);
- they assume the minimal number of internal states and the minimal number of flip-flops (practical examples show evidence that sometimes realizations with more than the minimal number of states and number of flip-flops are best because of their much simpler excitation and output functions);

- they are very laborious (only small sequential machines can be effectively processed with them);

- they give no information about the degree of optimality, for calculated assignments;

- they are not general in the sense that some machines have too few useful partition pairs.

The column based approach [4] has the following disadvantages:

- the minimal numbers of internal states and of memory elements are assumed;

- partitions are evaluated and selected for state assignment based on the observation of only a few very general factors that have an influence on the quality of the resulting assignment, such as: for all present states adjacent zero (one) entries for a given state variable in the columns for two codewise adjacent inputs in the assigned next-state table; all zero (one) entries for a given state variable in the whole column of the assigned next-state table; independence of a given state variable on all other state variables for a given value of the input; the possibility of obtaining some common terms;

- the method is laborious because all the proper partitions for the set of machine states must be constructed and evaluated - the number of these partitions grows rapidly with |S| (only small sequential machines can be effectively processed with it).

The method presented here is related to the third group of methods. Some of the observations, on which our method is based, are like those used by Armstrong [1][2]. However, many important differences exist between the method presented here and Armstrong's method.

Armstrong's method does not take into account:
the adjacency conditions that follow from the state-output dependencies;  
- the adjacency conditions that follow from the present-state–next-state dependencies, other than "the same next-state for two adjacent present-states" (type II adjacency [1]);  
- the adjacency conditions that follow from the input-state dependencies, other than "the adjacent next-states for a given present-state and two codewise adjacent inputs" (type I adjacency [1]).

The method of maximal adjacencies presented here considers all these sorts of adjacencies and the types of adjacencies specified by Armstrong are present implicitly among all the types of adjacency considered.

Thus, the method of maximal adjacencies uses much more information about the factors, that can have influence on the quality of the resultant assignments than the method of Armstrong [1][2] and, also, than all the other related methods [3][15][20][21]. Therefore, in many cases, it can produce better assignments than the methods of the third group. For the same reason, it can give better assignments than the column based approach.

The method presented here uses adjacency conditions, that are ordered according to the number of adjacencies reached when a given condition is satisfied by the assignment. The number of adjacencies reflects the condition's quality. Since the conditions are considered and combined, starting with the best, the first assignments constructed will be always nearly optimal and, almost always, the best of the nearly optimal solutions will be one of the first to be obtained.

In the methods using minimization of multi-valued, multi-output non-univocal functions for creating the conditions used further in order to construct the near optimal assignments, such a measure of quality of conditions and appropriate ordering relation on the set of conditions did not exist. Thus, the method of maximal adjacencies seems to be most effective, i.e. it produces good results more quickly. The capacity of this method is very important, especially, for large machines with "difficult" algebraic structures, for which the construction of assignments is time and memory consuming.
The method does not assume minimum numbers of states and memory elements and allows concurrent state assignment and (partial) minimization.

Furthermore, some of the best types of flip-flops can be adopted in order to realize each excitation function and the complexity of the realization of the output function is taken into account.

Finally, like the methods of the third group, the method of maximal adjacencies allows us to assign the states for large sequential machines. This ability results from two facts:
- neither the set of SP-partitions nor the partitions from I-S and S-O partition pairs need to be considered (the partition theory approach) and the set of proper partitions does not have to be considered (the column based approach). They grow quickly with the number of states of the machine. Here, the set of adjacency conditions has to be constructed and considered. This set contains only $|S| \cdot (|S|-1)/2$ conditions (for a machine with $|S|=100$ internal states, this requires only 4950 conditions);
- not all the possible combinations of adjacency conditions need to be considered, because the first combinations are always nearly optimal and very often, the best of them is obtained very early on.

The method of maximal adjacencies contains none of the shortcomings of the first two state assignment approaches.
**LITERATURE**

A programmed algorithm for assigning internal codes to sequential machines.

On the efficient assignment of internal codes to sequential machines.

Concurrent minimization and state assignment of finite state machines.

The coding of internal states of sequential circuits.

An efficient state minimization algorithm for some special classes of incompletely specified sequential machines.

[6] Friedman, A.D. and P.R. Menon
Theory and design of switching circuits.
Digital system design series.

[7] Hartmanis, J.
On the state assignment problems for sequential machines I.

[8] Stearns, R.E. and J. Hartmanis
On the state assignment problem for sequential machines II.

[9] Hartmanis, J.
Loop-free structure of sequential machines.

[10] Hartmanis, J.
Further results on the structure of sequential machines.

Some dangers in state reduction of sequential machines.
Pair algebra and its application to automata theory.  

Algebraic structure theory of sequential machines.  
Prentice-Hall series in automatic computation

[14] Hill, F.J. and G.R. Peterson  
Introduction to switching theory and logical design. 2nd ed.  

[15] Humphrey, Jr., W.S.  
Switching circuits with computer applications.  

[16] Karp, R.M.  
Some techniques of state assignment for synchronous sequential machines.  

[17] Lala, P.K.  
An algorithm for the state assignment of synchronous sequential circuits.  

[18] Lewin, D.  
Design of logic systems.  

Introduction to VLSI systems.  
Reading, Mass.:Addison-Wesley, 1981.  
Addison-Wesley series in computer science

[20] De Micheli, G. and G. Sangiovanni-Vincentelli, T. Villa  
Computer-aided synthesis of PLA-based finite state machines.  

[21] De Micheli, G. and R.K. Brayton, A. Sangiovanni-Vincentelli  
Optimal state assignment for finite state machines.  

[22] Noe, P.S. and V.T. Rhine  
Optimum state assignment for the D flip-flop.  

[23] Papachristou, Ch.A. and D. Sarma  
An approach to sequential circuit construction in LSI programmable arrays.  
[24] Pottosin, Yu.V.
Decompositional method for coding the states of a parallel automaton.

A state assignment technique for sequential machines using J-K flip-flops.

[26] Yamamoto, M.
A method for minimizing incompletely specified sequential machines.
Jóźwik, J.
THE FULL DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE STATE AND OUTPUT BEHAVIOUR REALIZATION.

Pineda de Gyvez, J.
ALWAYS; A system for wafer yield analysis.

Siuzdak, J.
OPTICAL COUPLERS FOR COHERENT OPTICAL PHASE DIVERSITY SYSTEMS.

Bastiaans, M.J.
LOCAL-FREQUENCY DESCRIPTION OF OPTICAL SIGNALS AND SYSTEMS.

Worm, S.C.J.
A MULTI-FREQUENCY ANTENNA SYSTEM FOR PROPAGATION EXPERIMENTS WITH THE OLYMPUS SATELLITE.

Kersten, W.F.J. and G.A.P. Jacobs
ANALOG AND DIGITAL SIMULATION OF LINE-ENERGIZING OVERVOLTAGES AND COMPARISON WITH MEASUREMENTS IN A 400 kV NETWORK.

Hosselet, L.M.L.F.
MARTINUS VAN MARUM: A Dutch scientist in a revolutionary time.

Bondarev, V.N.
ON SYSTEM IDENTIFICATION USING PULSE-FREQUENCY MODULATED SIGNALS.

Liu Wen-Jiang, Zhu Yu-Cai and Cai Da-Wei
MODEL BUILDING FOR AN INGOT HEATING PROCESS; Physical modelling approach and identification approach.

Liu Wen-Jiang and Ye Dau-Hua
A NEW METHOD FOR DYNAMIC HUNTING EXTREMUM CONTROL, BASED ON COMPARISON OF MEASURED AND ESTIMATED VALUE.

Liu Wen-Jiang
AN EXTREMUM HUNTING METHOD USING PSEUDO RANDOM BINARY SIGNAL.

Jóźwik, L.
THE FULL DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE OUTPUT BEHAVIOUR REALIZATION.

Huis in 't Veld, R.J.
A FORMALISM TO DESCRIBE CONCURRENT NON-DETERMINISTIC SYSTEMS AND AN APPLICATION OF IT BY ANALYSING SYSTEMS FOR DANGER OF DEADLOCK.

Woudenberg, H. van and R. van den Born
HARDWARE SYNTHESIS WITH THE AID OF DYNAMIC PROGRAMMING.

Engelshoven, R.J. van and R. van den Born
COST CALCULATION FOR INCREMENTAL HARDWARE SYNTHESIS.

Delissen, J.C.M.
THE LINEAR REGRESSION MODEL: Model structure selection and biased estimators.

Kalasek, V.K.I.
COMPARISON OF AN ANALYTICAL STUDY AND EMTP IMPLEMENTATION OF COMPLICATED THREE-PHASE SCHEMES FOR REACTOR INTERRUPTION.
(205) Butterweck, H.J. and J.H.F. Ritzerfeld, M.J. Werter
FINITE WORDLENGTH EFFECTS IN DIGITAL FILTERS: A review.

(206) Bollen, M.H.J. and G.A.P. Jacobs
EXTENSIVE TESTING OF AN ALGORITHM FOR TRAVELLING-WAVE-BASED DIRECTIONAL DETECTION AND PHASE-SELECTION BY USING TWINFIL AND EMTF.

(207) Schuurman, W. and M.P.H. Weenink
STABILITY OF A TAYLOR-RELAXED CYLINDRICAL PLASMA SEPARATED FROM THE WALL BY A VACUUM LAYER.

(208) Lucassen, F.H.R. and H.H. van de Ven
A NOTATION CONVENTION IN RIGID ROBOT MODELLING.

(209) Józwiak, L.
MINIMAL REALIZATION OF SEQUENTIAL MACHINES: The method of maximal adjacencies.

(210) Lucassen, F.H.R. and H.H. van de Ven
OPTIMAL BODY FIXED COORDINATE SYSTEMS IN NEWTON/EULER MODELLING.

(211) Boom, A.J.J. van den
Kw CONTROL: An exploratory study.