MINAS : an algorithm for systematic state assignment of sequential machines : computational aspects and results

tu/e

Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

MINAS: An Algorithm for Systematic State Assignment of Sequential Machines - Computational Aspects and Results

by
J.L. Duarte

EUT Report 89-E-217
ISBN 90-6144-217-6

April 1989

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

MINAS:
An algorithm for systematic state assignment of sequential machines - computational aspects and results

by

J.L. Duarte

EUT Report 89-E-217
ISBN 90-6144-217-6

Eindhoven
April 1989

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEE, DEN HAAG

Duarte, J.L.

MINAS: an algorithm for systematic state assignment of sequential machines - computational aspects and results / by J.L. Duarte. - Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering. - Tab. - (EUT report, ISSN 0167-9708; 89-E-217)
Met lit. opg., reg.
ISBN 90-6144-217-6
SISO 664 UDC 681.325.65:519.6 NUGI 832
Trefw.: automatentheorie.

MINAS :

an algorithm for systematic state assignment of sequential machines - Computational aspects and Results.

J.L. Duarte

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

Abstract- One of the central problems in the physical realization of sequential machines is the selection of binary codes to represent the internal states of the machine. The Method of Maximal Adjacencies can be viewed as an approach to the state assignment problem.

This research report concentrates on simple, practical strategies to implement that method.

A fully-described program in Pascal has been included and serves a two-fold purpose: (1) it exposes concrete practical solutions,which encourages the reader to try other strategies on his(her) own; (2) it has been conceived from a general standpoint that allows to check the correctness of different theoretic concepts emerging from the Method of Maximal Adjacencies.

A set of industrial sequential machines has been chosen to test the program. Results from other existing methods have been also reported.

Index terms- Automata theory, logic minimisation, logic system design, sequential machines.

Acknowledgements- We particulary thank Prof.ir. M.P.J. Stevens for making it possible to perform this work, dr.ir. L. Jóźwiak for his advice in the development of the program, and mr. C. van de Watering for having typed major parts of this report.

CONTENTS

I. INTRODUCTION

The Method of Maximal Adjacencies [1] is a new approach to attack some aspects related to an old problem : the minimal realization of sequential machines.

This report assumes that you are familiar with the theoretic concepts developed in [1]. Therefore, section III provides just a "down-to-earth" description of the central ideas in that method. A glossary is available in section II, where some terminology is introduced in a very informal way. We encourage you to consult [1] to be acquainted with formal definitions.

The purpose of the guide in section IV is to help you explore program MINAS presented in the next section. The guide has been organised to clarify the contents of the procedures following a stepwise refinement method. We advise you to read interactively the guide and the program.

Section V pays attention to the application program which implements major aspects of the Method of Maximal Adjacencies. It is described in the programming language Pascal. This program has been conceived to run on a "Apollo† computer/Domain† system" implementation. Great care has been taken to avoid implementation-dependent features of Pascal (the implementation-dependent aspects of the program are not essential to understand the algorithm).

As an illustration of the capabilities of the Maximal Adjacencies approach, results of some experiments have been included in section VI. The same set of industrial finite state machines has been used to test the program KISS, as reported in [2]. KISS is also a program for state encoding of sequential machines, based on a multi-valued, multi-output, non-univocal function minimization method. Results obtained by MINAS are compared with those ones obtained by KISS.

Finally this research report is finished with some conclusions and propositions for future works.

APOLLO and DOMAIN are registered trademarks of Apollo Computer Inc.

II. GLOSSARY

* Adjacency : two binary sequences (e.g. two input codes, two state codes, two product terms) of the same length are defined to be "adjacent" if the number of positions in which they differ is only one.

* Partition : Let S be the set of states of any finite state machine. A "partition" on S is a set of disjoint subsets of S whose set union is S .

* Block : an element of a partition. Two blocks of the same partition are always disjoint subsets.

* Final Family of Partitions (FFP) : every set of two_block partitions related to a finite state machine M, satisfying the following conditions:

  < i> the number of partitions within the FFP is equal to k, where 2**(k-1) < |S| <= 2**k and |S| is the number of states of M.

  < ii> each state is separated from each other in at least one partition; i.e., they are placed separated blocks in at least one partition.

  <iii> the number of elements in each block is less than or equal to 2**(k-1).

* Adjacent States : two states within a FFP which are in separated blocks only once .

  Example :

  Consider a finite state machine with 5 states. Then,

```
ffp1 = (  { [ 1 2 4 ] ; [ 3 5 ] }  ,
          { [ 2 3 4 5 ] ; [ 1 ] }  ,
          { [ 1 4 5 ] ; [ 2 3 ] }  )
```

  is a final family of partitions. State 3 and state 5 are adjacent. State 3 and state 4 are not adjacent.

  State assignment resulting from ffp1 :

```
State 1 ::  0 1 0
State 2 ::  0 0 1
State 3 ::  1 0 1
State 4 ::  0 0 0
State 5 ::  1 0 0
```

* State Pair: two non negative integers representing two different (incompatible) states of a finite state machine. The first state in the pair is supposed to be smaller than the second one.

* State Pair Position : for a given finite state machine, there is a series of ordered pairs of states where each pair can be uniquely identified by its position in the series.

  Example :

  Consider a finite state machine with 4 states. Then,

< i>	(1,2)	(1,3)	(1,4)	(2,3)	(2,4)	(3,4)
<ii>	1	2	3	4	5	6

  < i> series of state pairs
  <ii> state pair position in the series.

* Pattern : a symbol representing binary sequences. For instance, the input pattern (1 2 2 0) resumes the following input sequences: (1 0 0 0), (1 0 1 0), (1 1 0 0), (1 1 1 0). (don't care bits are represented by the integer "2").

* MNSC : Maximal Number of Simultaneously satisfied adjacency Conditions for a given set of state pairs.

* MNP : estimated average number of partitions containing a specified state pair at the same block within a final family of partitions.

* Cost : parameter instructing the number of state pairs, from a set of pairs, that should be actually made codewise adjacent.

III. CENTRAL IDEAS IN THE METHOD OF MAXIMAL ADJACENCIES

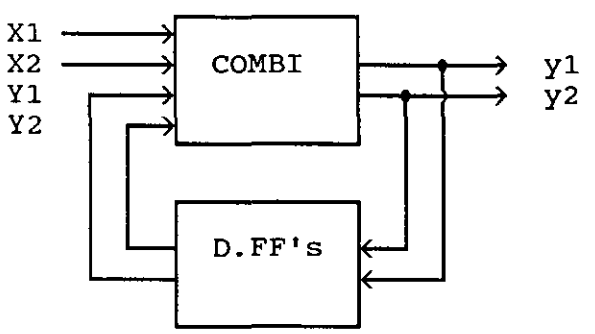
Consider the sequential machine with next-state table described at Table III.1 .

Table III.1

** Next State Table for Finite State Machine M **

I / PS	i0 00	i1 01	i3 11	i2 10
1	2	4	1	4
2	4	2	3	2
3	1	4	1	3
4	3	2	2	1

Fig. III.1 illustrates a sketch for a possible physical realization of this finite state machine.



X1 \ X2	0	1
0	i0	i1
1	i2	i3

Fig III.1 Physical realization of machine M.

The problem we face now is to choose "desirable" (binary) codes to represent the internal states of the machine. Most often "desirable" means fewest number of components in the resulting realization of COMBI (a combinational logic circuit).

Binary codes which reduce the functional dependence between the state variables lead to simpler logical equations for the Boolean function representing COMBI. In other words, we have to look for a minimal set of product terms for the purpose of reducing the complexity of the combinational circuit.

State encoding can be formulated as a matter of finding a final family of partitions for a given sequential machine.

The Method of Maximal Adjacencies allows us to construct final families of partitions leading to near optimal state encodings. The basic idea is as follows:

Let us come back to the machine M. Choosing at random a final family of partitions for this machine, say

({ [1 2] ; [3 4] },
{ [1 3] ; [2 4] })

results the state assignment illustrated at Table III.2
(notice that states in first blocks receive "0" as code, and positioning at second blocks implies "1" as code).

It is easy to see that line adjacencies at Table III.2(c) and Table III.2(d) leads to the possibility of reduction of the number of product terms (and the number of variables in each product term) in the "sum of products" related to the Boolean variables y1 and y2.

The adjacencies have four different origins:

(i) Adjacencies concerning input codes independent on position of states within the partitions.

For instance, it can be seen from Table III.2(a) that there is always a possible adjacency for y1 and for y2 between lines <14> and <15> at Table III.2(c).

(ii) Adjacencies concerning input codes depending upon position of states within the partitions.

For y2, the adjacency between lines <5> and <6> at Table III.2(c) is due to the fact that state 2 and state 4 are in the same block at the second partition.
But, this possibility could have been forecasted, before performing the state assignment, directly from Table III.2(a).

Try to convince yourself it is possible to assure from Table III.2(a) that, every time state 1 and state 4 are in the same block within the partitions, there will be 3 adjacencies at Table III.2(c) (between lines <3> and <4>, lines <9> and <10>, lines <10> and <11>).

(iii) Adjacencies concerning state codes, unconditional to block placement of next-states.

For instance, adjacency between lines <5> and <7> at Table III.2(d) was reached because (present)state 1 is codewise adjacent to (present)state 3 (in fact, 2 possible adjacencies are to be considered : one for y1 and one for y2). This situation could also have been foreseen from Table III.2(b).

Can you see from lines <14> and <15> at Table III.2(b) that, if state 2 were codewise adjacent to state 3, there would be always one possible adjacency ? (now, or one adjacency for y1, or one adjacency for y2, why?)

(iv) Adjacencies concerning state codes, conditional to block placement of next-states.

There is an adjacency for y1 between lines <11> and <12> at Table III.2(d) because : (a) (present)states 3 and 4 are codewise adjacents and (b) (next)states 1 and 2 are in the same block within the first partition.

From Table III.2(b) you can say that, for instance, observing lines <7> and <8>, every time the (next) states 2 and 4 are in the same block within the partitions there will be one possible adjacency at Table III.2(d) if (present)states 3 and 4 are codewise adjacent.

An analogous reasoning is applicable to output tables.

The Method of Maximal Adjacencies has been developed based upon the observation that the information comprised in the next-state and output tables of sequential machines instructs input-state, present-state--next-state and output-state dependencies for adjacency conditions.
Hence, different sorts of adjacency conditions can be combined for the purpose of ordering a list of pair of states.
Then, trying to induce a maximal number of adjacencies, final families of partitions can be filled up with those ordered pair of states, which are supposed to be codewise adjacent.
Therefore, the logical dependence between Boolean variables resulting from the state assignment is considerably reduced.

Table III.2

** Example of state encoding for machine M **

input	present state	next state
i0	1	2
i1		4
i3		1
i2		4
i0	2	4
i1		2
i3		3
i2		2
i0	3	1
i1		4
i3		1
i2		3
i0	4	3
i1		2
i3		2
i2		1

(a)

input		present state		next state	
X1	X2	Y1	Y2	y1	y2
0	0	0	0	0	1
0	1	0	0	1	1
1	1	0	0	0	0
1	0	0	0	1	1
0	0	0	1	1	1
0	1	0	1	0	1
1	1	0	1	1	0
1	0	0	1	0	1
0	0	1	0	0	0
0	1	1	0	1	1
1	1	1	0	0	0
1	0	1	0	1	0
0	0	1	1	1	0
0	1	1	1	0	1
1	1	1	1	0	1
1	0	1	1	0	0

(c)

input	present state	next state
i0	1	2
	2	4
	3	1
	4	3
i1	1	4
	2	2
	3	4
	4	2
i3	1	1
	2	3
	3	1
	4	2
i2	1	4
	2	2
	3	3
	4	1

(b)

input		present state		next state	
X1	X2	Y1	Y2	y1	y2
0	0	0	0	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	0	1	0	1
0	1	1	0	1	1
0	1	1	1	0	1
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	0	0

(d)

(a),(b) : other representation for next-state table.
(c),(d) : encoding for machine M.

IV. GUIDE TO MINAS

Cross-references have been used within the guide, which means that it contains a certain amount of duplication, but this was accepted on the ground that the repetitions in their context contribute to a better understanding of the characteristics of the program under discussion.

PREVIEW :

1. MINAS

Algorithm based on the Method of Maximal Adjacencies for systematic encoding of Finite State Machines (FSM).

The algorithm looks for a minimal structure, based upon D-flip-flop's as memory elements, which realises the same input/output behaviour as the FSM described in the input file.

(a) OVERVIEW :

1. MINAS

{ begin }

1.1 GetReferenceTime;

1.2 GetPrimitiveData;

1.3 ComputeInputStateAdjacencyConditions;

1.4 ComputeOutputStateAdjacencyConditions;

1.5 ComputePresentStateNextStateAdjacencyConditions;

1.6 SortPriorities;

1.7 GenerateFinalFamiliesOfPartitionsBasedOnPriorities;

1.8 RecordBestFinalFamilyOfPartitions;

1.9 RecordLoopTime;

{ end }

(b) HIGHLIGHTS :

* 1.1 GetReferenceTime : The "Domain" system provides a number of system routines to manipulate time.
Information about the ways the system represents time, how to get time from the system, and how to manipulate time you can find in the " Apollo Domain - Programming With General System Calls " manual.
Procedure GetReferenceTime just uses system routines to assign to the variable ref_sec the starting execution time from MINAS.
Later this reference time will be used to give information about global execution time.

* 1.2 GetPrimitiveData : The input data file format definition for MINAS is given in Appendix 1.
Internaly in the algorithm this file is called PrimitiveData. Notice that don't care inputs or outputs are represented by the integer "2"; present states receive positive integers as symbols, and next states are represented by non negative integers (next state = 0 means that next state is "don't care"!).

Based upon PrimitiveData, the purpose of the procedure GetPrimitiveData is fill up the following arrays:

(i) InputMatrixDef : every input pattern receives a identifier (a non negative integer).

(ii) OutputMatrixDef: every output pattern receives a identifier (a non negative integer).

(iii) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(iv) OutputTable : as entries to this table you have an input pattern identifier and a present state ; as output you have an output pattern identifier. (output pattern identifier = 1 stands for don't care output at every output line).

* 1.3 ComputeInputStateAdjacencyConditions : After getting information about the FSM structure,
MINAS has to recognise input/present-state adjacency conditions. The algorithm handles with every possible input subspace by expanding input symbols.

Based on the results from GetPrimitiveData, this procedure aims to fill up the following arrays :

(i) VectorInputState : number of possible adjacencies for each state pair depending upon the inputs to the FSM.

(ii) VectorInputCorrelation : number of same don't care bit positions considering two different input patterns.

(iii) VectorInputAutoCorrelation : number of don't care bits in an input pattern.

* 1.4 ComputeOutputStateAdjacencyConditions : identify possible adjacency conditions due to the structure of the different output patterns.
The following array is constructed by this procedure:

(i) VectorOutputState : number of possible adjacencies for each state pair due to the FSM outputs.

* 1.5 ComputePresentStateNextStateAdjacencyConditions: besides calculating present-state-- next- state adjacency conditions, this procedure also combines other adjacency conditions and gives an estimation of the total number of adjacencies for each state pair. Only the necessary information is stored for future use.

Arrays resulting from this procedure:

(i) VectorLineAux : for a given state pair, this vector contains combined adjacencies conditional to block placement, taking into account present state--next-state and input-state dependencies.

(ii) VectorDontCare : there are some present state--next-state adjacencies independent from next state block placement.

(iii) VectorUnconditional : all possible adjacencies not dependent from next state block placement.

(iv) ConditionalTable : for each state pair there is an ordered series of state pairs resulting on a maximal number of adjacencies that depends on block placement. The series' lenght is imposed by the parameter MNSC (maximal number of simultaneously satisfied adjacency conditions).

(v) ConditionalAdjacencies :in ConditionalTable you have information about conditional state pairs. The number of conditional reached adjacencies concerning each conditional state pair will be found in the table ConditionalAdjacencies.

(vi) VectorEstimationTotal: estimation of the total number of adjacencies if a given state pair has been codewise adjacent.
This estimation depends upon the parameter MNP (estimated average number of partitions containing a specified state pair at the same block).

* 1.6 SortPriorities : this procedure orders priorities aiming to build a final family of partitions.
Retrieving information from VectorEstimationTotal, the following array will be constructed :

(i) VectorPriority: ordered series of state pairs supposed to have preferencial entry while building a family of partitions.

* 1.7 GenerateFinalFamiliesOfPartionsBasedOnPriorities : Final families of partitions are built sequentially obeying priorities.
Here a "branch and bound" approach has been used for the purpose of placing states into partition blocks.
Once finished, the current Final Family of Partitions (FFP) has to be shaped in order to allow state encoding. After that, the results are sent to a file which will be read by the minimiser.
MINAS has to wait for the answer comming from the minimiser in order to proceed. Depending on the achieved number of product terms, the algorithm decides either to go on or not.

* 1.8 RecordBestFinalFamilyOfPartitions : after having decided for the "best" final family of partitions (that is, the assignment resulting the smallest number of product terms), MINAS registers the results in a file matching the minimiser input format definition.
(see Appendix 1).

* 1.9 RecordLoopTime : same remarks as for GetReferenceTime (1.1).
This procedure calculates and shows the algorithm's global execution time, including the required time for minimisation.

(c) INPUTS :

(i) input file containing FSM symbolic description (Appendix 1).

(ii) file with results from minimiser (Annexe 1).

(d) OUTPUTS :

(i) output file containing encoded FSM description, matching minimiser input file format definition (Annexe 1).

(e) TOOLS :

(i) Push : procedure intented to place a new component into a stack of non negative integers.

(ii) Pop : procedure intented to take back the value of the latest component from a stack of non negative integers.

(iii) PushVector: procedure intented to place a new component into a stack of vectors.

(iv) PopVector : procedure intented to take back the value of the latest component from a stack of vectors.

(v) Order : this procedure examines and, if necessary, exchanges the values of two states A and B, so that the value of A is smaller than the value of B.

(vi) Increment : the goal of this procedure is to add "one" to a given non negative integer.

(vii) AddTo : adds a non negative integer to a real variable.

(viii) Power : calculates 2 raised to a given power.

(ix) Field : returns field format for printing integers.

(x) Locate: this procedure receives a state pair and gives back its respective position in the series of ordered pair of states.

(xi) Decode: this procedure gets a state pair position in the series of ordered pairs of states and gives back the constitutive states.

(xii) DecodeInp : this procedure gets a input pair position in the series of ordered pairs of inputs and computes the constitutive input pattern identifiers.

(xiii) QuickSort : this is the "quick sort algorithm" for sorting variables in an array . The previous state pair positions are also preserved in an other array.

(xiv) ShortQuickSort : the same as QuickSort, but the concerned arrays are shorter in size.

1.1 GetReferenceTime

The "Domain" system provides a number of system routines to manipulate time.
Information about the ways the system represents time, how to get time from the system, and how to manipulate time you can find in the " Apollo Domain - Programming With General System Calls " manual.
Procedure GetReferenceTime just uses system routines to assign to the variable ref_sec the starting execution time from MINAS.
Later this reference time will be used to give information about global execution time.

1.2 GetPrimitiveData

The input data file format definition for MINAS is given in Appendix 1. Internaly in the algorithm this file is named PrimitiveData. Notice that don't care inputs or outputs are represented by the integer "2"; present states receive positive integers as symbols, and next states are represented by non negative integers (next state = 0 means that next state is "don't care"!).

(a) OVERVIEW :

```
1.2 GetPrimitiveData

    { begin }

      { set initial conditions };

      { open file for receiving information };

      { get data and built arrays };

      { close file };

    { end };
```

(b) INPUTS :

(i) input file containing FSM symbolic description (Appendix 1).

(c) OUTPUTS :

(i) InputMatrixDef : every input pattern receives a identifier (a non negative integer).

(ii) OutputMatrixDef: every output pattern receives a identifier (a non negative integer).

(iii) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(iv) OutputTable : as entries to this table you have an input pattern identifier and a present state ; as output you have an output pattern identifier. (output pattern identifier = 1 stands for don't care output at every output line).

1.3 ComputeInputStateAdjacencyConditions

After getting information about the FSM structure, MINAS has to recognise input/present-state adjacency conditions.
The algorithm handles with every possible input subspace by expanding input symbols.

(a) OVERVIEW :

```
1.3 ComputeInputStateAdjacencyConditions

  { begin }

    { set initial conditions };

    { check input pattern binary correlation; i.e.
      the number of common don't cares between input
      patterns };

    { compute input_state adjacency conditions for every
      state pair : }

      { repeat : }

        { specify PresentState ( a state ) };

        { for every pair of input symbols do,
          if exist next states related to the current
          PresentState (say NextStateA and NextStateB)};

          { test if next state pair has already been
            used; if not ,then : }

            1.3.1 FullInInputStateCoincidenceMatrix;

            1.3.2 GenerateAllVirtualSubspaces;

            1.3.3 KeepOnlyActualSubspaces;

            { Adjacencies := 0 };

            1.3.4 DetectPossibleInputStateAdjacencies;

            { store number of maximal possible adjacen-
              cies for the current state pair };

        { end do }

      { until all states have been considered };

  { end };
```

(b) HIGHLIGHTS :

* 1.3.1 FullInInputStateCoincidenceMatrix : the inputs for this procedure are :

(i) PresentState : (reference state);

(ii) NextStateA : (state from pair under focus);

(iii) NextStateB : (state from pair under focus);

(iv) NextStateTable: as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

As outputs you have:

(i) CoincMatrixA : array of input symbols which are related to PresentState and NextStateA.

(ii) CoincMatrixB : array of input symbols which are related to PresentState and NextStateB.

* 1.3.2 GenerateAllVirtualSubspaces : for every possible pair of input patterns that can be extracted from CoincMatrixA and CoincMatrixB (one symbol from each array, different from don't care definition) there is a related subspace from which the pair of states NextStateA/NextStateB is a member. If in this (virtual) subspace no other states are present, then a real subspace is achieved allowing high order adjacencies.

List resulting from this procedure:

(i) SubspacesWaitingList : list of vectors defining possible subspaces.

* 1.3.3 KeepOnlyActualSubspaces : not every subspace definition stored in SubspacesWaitingList is a validy one. Only the subspaces containing the states NextStateA and NextStateB, and no other states, are of interest.
This procedure checks the subspaces and keeps the actual ones in the list :

(i) BookedSubspacesList : list of vectors defining actual subspaces.

* 1.3.4 DetectPossibleInputStateAdjacencies : the subspaces stored in Booked_SubspacesList are not necessarily mutualy exclusive ones; that means, one subspace can overlap the neighbouring. Hence, an optimal combination of mutually exclusive subspaces has to be chosen in order to result in a maximal number of adjacencies.

(c) INPUTS :

(i) InputMatrixDef : every input pattern receives a identifier (a non negative integer).

(ii) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(d) OUTPUTS :

(i) VectorInputState : number of possible adjacencies for each state pair depending on the inputs to the FSM.

(ii) VectorInputCorrelation : number of same don't care bit positions considering two different input patterns.

(iii) VectorInputAutoCorrelation : number of don't care bits in an input pattern.

1.3.1 FullInInputStateCoincidenceMatrix

(a) OVERVIEW :

1.3.1 FullInInputStateCoincidenceMatrix

{ begin }

{ look for input patterns resulting on the same next state as NextStateA or NextStateB for a given PresentState };

{ end };

(b) INPUTS :

(i) PresentState : (reference state);

(ii) NextStateA : (state from pair under focus);

(iii) NextStateB : (state from pair under focus);

(iv) NextStateTable: as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(c) OUTPUTS :

(i) CoincMatrixA : array of input symbols which are related to PresentState and NextStateA.

(ii) CoincMatrixB : array of input symbols which are related to PresentState and NextStateB.

1.3.2 GenerateAllVirtualSubspaces

For every possible pair of input patterns that can be extracted from CoincMatrixA and CoincMatrixB (one symbol from each array, different from don't care definition; see 1.3.1) there is a related subspace where the state pair NextStateA/NextStateB is present. If in this (virtual) subspace no other states are present then a real subspace is achieved allowing high order adjacencies.

(a) OVERVIEW :

```
1.3.2 GenerateAllVirtualSubspaces

      { begin }

        { repeat : }

          { choose two input patterns different from don't
            care definition; one from CoincMatrixA, the
            other from CoincMatrixB };

          1.3.2.1 RecursiveCreation;

        { until all possible pair of input patterns has
          been tried };

      { end };
```

(b) HIGHLIGHTS :

* 1.3.2.1 RecursiveCreation : looks for all possible subspaces that can be generated from the two input patterns under consideration. This procedure is auto-recursive and makes use of an internal procedure TestSequence to check the moment to stop the branched recursions. At the end of each recurrence a vector defining a subspace is stored.
For instance, the vector

(1 1 2 0 2)

defines a second-order subspace; and the vector

(1 2 2 2 2)

defines a fourth-order subspace ("2" stands for don't care bit).

(c) INPUTS :

(i) CoincMatrixA : array of input symbols which are related to PresentState and NextStateA.

(ii) CoincMatrixB : array of input symbols which are related to PresentState and NextStateB.

(d) OUTPUTS :

(i) SubspacesWaitingList : list of vectors defining possible subspaces.

1.3.3 KeepOnlyActualSubspaces

Not every subspace definition stored in SubspacesWaiting-List is a validy one. Only the subspaces containing the states NextStateA and NextStateB, and no other states, are of interest.
This procedure checks the previous defined subspaces and keeps the actual ones.

(a) OVERVIEW :

```
1.3.3 KeepOnlyActualSubspaces

     { begin }

       { repeat : }

         { get a subspace definition from Subspaces-
           WaitingList };

         1.3.3.1 FullInEigenVectorsSet;

         1.3.3.2 CheckBaseOrthogonality;

         { if orthogonality then store definition in
           BookedSubspacesList };

       { until every definition has been tried };

     { end };
```

(b) HIGHLIGHTS :

* 1.3.3.1 FullInEigenVectorsSet : that means, look for input symbols possibly matching the subspace definition under consideration. Hence, only input symbols that, for the current PresentState, imply in next states equal to NextStateA or NextStateB or don't care next state are the good ones.
Selected input symbols are stored in the list:

(i) EigenVectorsList.

* 1.3.3.2 CheckBaseOrthogonality : now we have to prove that the input patterns within EigenVectorsList fully generate the subspace definition under consideration; i.e. we have to check the Eigen vectors' "orthogonality".

(c) INPUTS :

(i) PresentState : (reference state);

(ii) NextStateA : (state from pair under focus);

(iii) NextStateB : (state from pair under focus);

(iv) InputMatrixDef : every input pattern receives a identifier (a non negative integer).

(v) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(vi) SubspacesWaitingList : list of vectors defining possible subspaces.

(d) OUTPUTS :

(i) BookedSubspacesList : list of vectors defining actual subspaces.

1.3.4 DetectPossibleInputStateAdjacencies

The subspaces stored in BookedSubspacesList are not necessarily mutualy exclusive ones; that means, the subspaces can overlap. Hence, an optimal combination of mutually exclusive subspaces has to be choosen in order to result in a maximal number of adjacencies.

(a) OVERVIEW :

```
1.3.4 DetectPossibleInputStateAdjacencies

      { begin }

        { create backup list from BookedSubspacesList
          aiming future information }

        { repeat }

          { get a subspace definition, say SubspaceA }

          { compare SubspaceA with other definitions and
            keep in ExclusiveList only the subspaces
            without overlapping  with SubspaceA }

          { update ExclusiveList; that means, there are
            possibly some subspaces within ExclusiveList
            which overlaps themselves. Keep only the
            mutualy exclusive  subspaces with biggest
            dimensions }

          { count possible number of reached adjacencies;
            i.e. the number of commun don't cares between
            SubspaceA and the other definitions in the
            updated ExclusiveList }

          { compare the number of possible adjacencies
            with the maximal reached until now; if
            necessary, update this last value.
            Hint : SubspaceA can be a  subset from other
            subspace definitions within BookedSubspaces-
            List;  that is why we have to update the
            maximal number of possible adjacencies }

        { until each subspace definition in BookedSubspa-
          cesList has been tried }

        { retrieve information in BookedSubspacesList }

      { end }
```

(b) INPUTS :

(i) BookedSubspacesList : list of vectors defining actual subspaces.

(c) OUTPUTS :

(i) Adjacencies : maximal number of possible adjacencies for a given pair of states.

1.4 ComputeOutputStateAdjacencyConditions

Identify possible adjacency conditions due to the structure of the different output patterns.

(a) OVERVIEW :

```
1.4 ComputeOutputStateAdjacencyConditions

    { begin }

      { set initial conditions };

      { for each state pair do : }

        { repeat : }

           1.4.1 CountPossibleOutputStateAdjacencies;

        { until each input pattern has been checked };

        { repeat : }

           1.4.1 CountPossibleOutputStateAdjacencies;

        { until each possible input pattern combination
          has been checked };

      { end do };

    { end };
```

(b) HIGHLIGTS :

* 1.4.1 CountPossibleOutputStateAdjacencies : this procedure compares bit by bit the different output patterns and updates the number of possible adjacencies due to the intercorrelation between states and outputs.
An internal procedure "CompareBitByBitOutput" returns information about the matching between bits from two different output patterns. This information is important in order to update the number of possible adjacencies related to output patterns.

(c) INPUTS :

(i) VectorInputCorrelation : number of same don't care bit positions considering two different input patterns.

(ii) VectorInputAutoCorrelation : number of don't care bits in an input pattern.

(iii) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(iv) OutputMatrixDef: every output pattern receives a identifier (a non negative integer).

(d) OUTPUTS :

(i) VectorOutputState : number of adjacency conditions for each state pair due to the FSM outputs.

1.5 ComputePresentStateNextStateAdjacencyConditions

Besides calculating present-state--next-state adjacency conditions, this procedure also combines other adjacency conditions and gives an estimation of the total number of adjacencies for each state pair. Only the necessary information is stored for future use.

(a) OVERVIEW :

```
1.5 ComputePresentStateNextStateAdjacencyConditions

    { begin }

      { set initial conditions };

      { for each state pair do }

        { reset initial conditions };

        { repeat : }

           1.5.1 CountPossiblePresStNextStAdjacencies;

        { until each input pattern has been checked };

        { repeat : }

           1.5.1 CountPossiblePresStNextStAdjacencies;

        { until each possible input pattern combination
          has been checked };

        { compute independently reached combined
          adjacencies };

        { compute combined adjacencies conditional to
          block placement, taking into account present-
          state--next-state and input-state dependencies};

        { sort number of combined conditional adjacencies
          taken into consideration the current results in
          VectorLineAux ( for a given state pair, this
          vector contains combined adjacencies conditional
          to block placement; that is, present-state--next-
          state and input-state dependencies). };

        { store in ConditionalTable and ConditionalAdja-
          cencies only the necessary information; i.e.
          a limited number of conditional pairs };

        { estimate the total number of adjacencies };

      { end do };

    { end };
```

(b) HIGHLIGHTS :

* 1.5.1 CountPossiblePresStNextStAdjacencies : there are some adjacencies that do not depend on next-states (this information is stored in VectorDontCare), and adjacencies that do depend on next-state block placement (this information is temporarily stored in VectorLineAux).

(c) INPUTS :

(i) NextStateTable: as entries to this table you have an input pattern identifier and a present-state ; as output you have a next-state (next-state = -1 means that the current input pattern is not related to the current present-state).

(ii) VectorInputState: number of adjacency conditions for each state pair depending on the inputs to the FSM.

(iii) VectorOutputState: number of adjacency conditions for each state pair due to the FSM outputs.

(iv) VectorInputCorrelation: number of same don't care bit positions considering two different input patterns.

(v) VectorInputAutoCorrelation: number of don't care bits in an input patterns.

(d) OUTPUTS :

(i) VectorDontCare : there are some present-state--next-state adjacencies independent from next-state block placement.

(ii) VectorUnconditional : all possible adjacencies not dependent from next state block placement.

(iii) ConditionalTable : for each state pair there is an ordered series of state pairs resulting on a maximal number of adjacencies that depends on block placement. The series' lenght is imposed by the parameter MNSC (maximal number of simultaneously satisfied adjacency conditions).

(iv) ConditionalAdjacencies :in ConditionalTable you have information about conditional state pairs. The number of conditional reached adjacencies concerning each conditional state pair will be found in the table ConditionalAdjacencies.

(v) VectorEstimationTotal : estimation of the total number of adjacencies if a given state pair has been codewise adjacent. This estimation depends on the parameter MNP (estimated average number of partitions containing a specified state pair at the same block).

1.6 SortPriorities

This procedure orders priorities aiming to build a final family of partitions.

(a) OVERVIEW :

1.6 SortPriorities

```
{ begin }

  { retrieve information form VectorEstimationTotal };

  { sort priorities };

  { store results in VectorPriority };

{ end };
```

(b) INPUTS :

(i) VectorEstimationTotal: estimation of the total number of adjacencies if a given state pair has been codewise adjacent. This estimation depends on the parameter MNP (estimated average number of partitions containing a specified state pair at the same block).

(c) OUTPUTS :

(i) VectorPriority: ordered series of state pairs supposed to have preferencial entry while building a family of partitions.

1.7 GenerateFinalFamiliesOfPartitionsBasedOnPriorities

Final families of partitions are built sequentially obeying priorities. Here a "branch and bound" approach has been used for the purpose of placing states into partition blocks.
Once finished, the current Final Family of Partitions (FFP) has to be shaped in order to allow state encoding. After that, the results are sent to a file which will be read by the minimiser.
MINAS has to wait for the answer comming from the minimiser in order to proceed. Depending on the achieved number of product terms, the algorithm decides either to go on or not.

(a) OVERVIEW :

```
1.7 GenerateFinalFamiliesOdPartitionsBasedOnPriorities

    { begin }

      { set initial conditions };

       { repeat : }

         { choose a state pair as priority head };

         1.7.1 BuiltFinalFamilyOfPartitions;

         1.7.2 ShapeFinalFamilyOfPartitions;

         1.7.3 RecordFinalFamilyOfPartitions;

         1.7.4 MinimizeMachineStructure;

         1.7.5 AnalyzeMinimizedStructure;

       { until optimal concept is reached };

    { end }
```

(b) HIGHLIGTS :

* 1.7.1 BuiltFinalFamilyOfPartitions : Two states are codewise adjacent if they are placed in separate blocks only at one partition of the FFP, and if placed together in all the other partitions.
Starting with the priority head, this procedure chooses state pairs and iserts them into the current FFP.
If one state of the choosen state pair is a state member, the other state is made codewise adjacent to it.
(State member means that the state has already been placed in all partitions).
If both states are not state members, one state is placed codewise adjacent to an old member; then, if possible, the second state is placed codewise adjacent to the first one.
This routine is followed until all states become members.

* 1.7.2 ShapeFinalFamilyOfPartitions : State assignment depends on state position within a partition (states in the first block are coded with "0", in the second block coded with "1").
In order to shape the "best" block positions, this procedure takes into consideration the number of entries of the states in the NextStateTable.

* 1.7.3 RecordFinalFamilyOfPartitions : after having built and shaped a FFP, the results are registered in a file matching the simplest version of the minimiser input format definition (Annexe 1).

* 1.7.4 MinimizeMachineStructure : MINAS has no internal procedure allowing minimisation of encoded FSM's.
In order to achieve this purpose, we have decided to use the library program MOM (Multiple Output Minimiser), available in the TUE-Lib.
For the purpose of information exchange between the two programs, the mailbox concept is needed. A mailbox is an object (a file) that two (or more) programs use to get messages or to put messages in.
To read or to write using mailboxes, MINAS uses the " MBX system calls" described in detail in the mailbox chapter of the "Domain - Programming With System Calls for Interprocess Communication" manual.

* 1.7.5 AnalyzeMinimizedStructure : The results from MOM (a minimized FSM) are available in a file (format definition given in Annexe 1).
MINAS gets data from this file in order to decide which assignment will be the best.

(c) INPUTS :

(i) VectorPriority : ordered series of state pairs supposed to have preferencial entry while building a family of partitions.

(ii) ConditionalTable : for each state pair there is an ordered series of state pairs resulting on a maximal number of adjacencies that depends on block placement. The series' lenght is imposed by the parameter MNSC.

(iii) ConditionalAdjacencies :in ConditionalTable you have information about conditional state pairs. The number of conditional reached adjacencies concerning each conditional state pair will be found in the table ConditionalAdjacencies.

(iv) NextStateTable : as entries to this table you have an input pattern identifier and a present-state ; as output you have a next-state (next-state = -1 means that the current input pattern is not related to the current present-state).

(v) file with results from the minimiser (Annexe 1).

(d) OUTPUTS :

(i) FFPKey : best final family of partitions.

1.7.1 BuiltFinalFamilyOfPartitions

Two states are codewise adjacents if they are placed in separate blocks only at one partition of the FFP, and if placed together in all the other partitions.
Starting with the priority head, this procedure chooses state pairs and iserts them into the current FFP.
If one state of the choosen state pair is a state member, the other state is made codewise adjacent to it.
(State member means that the state has already been placed in all partitions).
If both states are not state members, one state is placed codewise adjacent to an old member; then, if possible, the second state is placed codewise adjacent to the first one.
This routine is followed until all states become members.

(a) OVERVIEW :

```
1.7.1 BuiltFinalFamilyOfPartitions

      { begin }

        { get value for priority head };

        { set initial conditions };

        { place state pair identified by priority head in
          partitions of the FFP };

        { place conditional state pairs related to priority
          head in all paritions : }

          { ReferencePosition := PriorityHead }

          1.7.1.1 FullInVectorAuxiliarWithConditionalPairs;

          1.7.1.2 PlaceConditionalPairsFromVectorAuxiliar;


        { place other state pairs obeying priorities, until
          FFP is finished : }

          { repeat : }

            { get value for priority head auxiliar };

            { ConditionalPair := PriorityHeadAux };

            1.7.1.3 MakeLinkForStatePair;

            { if FFP not finished then }

              1.7.1.4 DetectNeighbouring;

              { if PriorityHeadAux codewise adjacents then}

                { ReferencePosition := PriorityHeadAux };
```

```
                    1.7.1.1 FullInVectorAuxiliarWithConditionalPairs;

                    1.7.1.2 PlaceConditionalPairsFromVectorAuxiliar;

                  { end if };

                { end if };

             { until FFP is finished };

         { end };
```

(b) HIGHLIGHTS :

* 1.7.1.1 FullInVectorAuxiliarWithConditionalPairs : for a given Reference-Position (that is, for a given state pair identifier), this procedure looks for conditional state pairs stored in the ConditionalTable in orther to place then into the FFP.
Conditional pairs are chosen following a prespecified weight which depends upon the parameter "Cost".

Array resulting from this procedure :

(i) VectorLocus : state pair identifiers from ConditionalTable related to Reference-Position and presenting more conditional adjacencies than the prespecified weight.

* 1.7.1.2 PlaceConditionalPairsFromVectorAuxiliar : each state pair from Vectorlocus will be placed, if possible, into the partition blocks.

* 1.7.1.3 MakeLinkForStatePair : Some requirements must be fulfilled in order to place a ConditionalPair into a FFP. An important boundary condition is the maximal allowed number of adjacencies for state member. (State member means that the state has been already placed in all partitions).
If one state of the chosen ConditionalPair is a state member, the other state is made codewise adjacent to it.
If both states are not state members, one state is placed codewise adjacent to an old member; then, if possible, the second state is placed codewise adjacent to the first one.

* 1.7.1.4 DetectNeighbouring : it is meaningful to try to place conditional state pairs related to the current PriorityHeadAux only if the state pair PriorityHeadAux is self codewise adjacent.
This procedure checks if the states from PriorityHeadAux are adjacents.

(c) INPUTS :

(i) PriorityHead : state pair which initialises the construction of a FFP.

(ii) VectorPriority : ordered series of state pairs supposed to have preferencial entry while building a family of partitions.

(iii) ConditionalTable : for each state pair there is an ordered series of state pairs resulting on a maximal number of adjacencies that depends on block placement. The series' lenght is imposed by the parameter MNSC.

(iv) ConditionalAdjacencies :in ConditionalTable you have information about conditional state pairs. The number of conditional reached adjacencies concerning each conditional state pair will be found in the table ConditionalAdjacencies.

(v) VectorUnconditional : all possible adjacencies not dependent from next state block placement.

(vi) Cost : parameter imposing the number of conditional state pairs which will be catched out the ConditionalTable for the purpose of built a FFP.

(d) OUTPUTS :

(i) FFP : final family of partitions resulting from PriorityHead.

1.7.1.1 FullInvectorAuxiliarWithConditionalPairs

For a given ReferencePosition (that is, for a given state pair identifier), this procedure selects conditional state pairs stored in the ConditionalTable in order to place then into the FFP. Conditional pairs are chosen following a prespecified weight which depends upon the parameter "Cost".

(a) GENERAL VIEW :

```
1.7.1.1 FullInVectorAuxiliarWithConditionalPairs

        { begin }

          { get information in ConditionalTable and
            in ConditionalAdjacencies related to
            ReferencePosition };

          { update number of adjacencies taking into
            account that now the state pairs will be
            actually codewise adjacents };

          { choose weight };

          { keep only state pairs presenting number of
            adjacencies greater than or equal to weight};

        { end };
```

(b) INPUTS :

(i) ReferencePosition : state pair conducting the choice of state pairs out the Conditional-Table.

(ii) ConditionalTable : for each state pair there is an ordered series of state pairs resulting on a maximal number of adjacencies that depends on block placement. The series' lenght is imposed by the parameter MNSC .

(iii) ConditionalAdjacencies :in ConditionalTable you have information about conditional state pairs. The number of conditional reached adjacencies concerning each conditional state pair can you find in the table ConditionalAdjacencies.

(iv) VectorUnconditional : all possible adjacencies not dependent from next state block placement.

(v) Cost : parameter imposing the number of conditional state pairs which will be catched out the ConditionalTable for the purpose of built a FFP.

(c) OUTPUTS :

(i) VectorLocus : state pair identifiers from ConditionalTable related to ReferencePosition and presenting more conditional adjacencies than the prespecified weight.

1.7.1.2 PlaceConditionalPairsFromVectorAuxiliar

If possible, each state pair from VectorLocus will be placed into the partition blocks.

(a) OVERVIEW :

```
1.7.1.2 PlaceConditionalPairsFromVectorAuxiliar

        { begin }

          { set initial conditions };

          { repeat : }

            { get a state pair from VectorLocus }

            1.7.1.3 MakeLinkForStatePair;

          { until VectorLocus is exhausted
                or FFP is finished };

        { end };
```

(b) INPUTS :

(i)	VectorLocus	: state pair identifiers from ConditionalTable related to ReferencePosition and presenting more conditional adjacencies than the pre-specified weight.
(ii)	ReachedAdjacencies	: array containing the current number of reached adjacencies for every state.
(iii)	SetOfAdjStates	: array containing the set of current adjacent states to every state.
(iv)	FFPStateMembership	: set with current state members.
(v)	FFP	: (current status)

(d) OUTPUTS :

(i)	ReachedAdjacencies	: (updated)
(ii)	SetOfAdjStates	: (updated)
(iii)	FFPStateMembership	: (updated)
(iv)	FFP	: (updated)

1.7.1.3 MakeLinkForStatePair

Some requirements must be fullfilled in order to place a ConditionalPair into a FFP. An important boundary condition is the maximal allowed number of adjacencies for state member. (State member means that the state has been already placed in all partitions).
If one state of the chosen ConditionalPair is a state member, the other state is made codewise adjacent to it.
If both states are not state members, one state is placed codewise adjacent to an old member; then, if possible, the second state is placed codewise adjacent to the first one.

(a) OVERVIEW :

```
1.7.1.3 MakeLinkForStatePair

        { begin }

          { state pair under consideration is identified
            by the variable ConditionalPair };

          { get current reached adjacencies for states
            identified by ConditionalPair };

          { if one state is a state member owning less
            adjacencies than the maximal allowed and the
            other state is not a member  then : }

             1.7.1.3.1 StoreStatePair;

          { else : }

            { if both states are not state members then: }

              { introduce the first state : }

                  1.7.1.3.2 LookForOverlapping;

                  1.7.1.3.1 StoreStatePair;

              { if reached adjacencies for first state is
                less than allowed then : }

                { introduce second state adjacent to the
                  first one : }

                    1.7.1.3.1 StoreStatePair;

              { else :  introduce second state adjacent to
                other state member : }

                  1.7.1.3.2 LookForOverlapping;

                  1.7.1.3.1 StoreStatePair;

            { else : }

              { if one state's reached adjacencies is the
                maximal allowed , and the other state is
                not a state member then : }
```

```
                    { introduce second state adjacent to any
                      other state member : }

                        1.7.1.3.2 LookForOverlapping;

                        1.7.1.3.1 StoreStatePair;

                  { else : both states are state members;
                           do nothing }

          { end };
```

(b) HIGHLIGHTS :

* 1.7.1.3.1 StoreStatePair : first, the state pair is placed together in the blocks of all partitions; after, the states under consideration are put apart in only one partition, that means they are made adjacent states.
Finally the number of reached adjacencies for every state member is updated.

* 1.7.1.3.2 LookForOverlapping : this procedure looks for a state member owning less reached adjacencies than the maximal allowed, for the purpose of introducing a new state in the FFP adjacent to it .

(c) INPUTS :

(i) ConditionalPair : state pair supposed to be inserted into the FFP.

(ii) ReachedAdjacencies: array containing the current number of reached adjacencies for every state.

(iii) SetOfAdjStates : array containing the set of current adjacent states to every state.

(iv) FFPStateMembership: set with current state members.

(v) FFP : (current status)

(d) OUTPUTS :

(i) ReachedAdjacencies : (updated)
(ii) SetOfAdjStates : (updated)
(iii) FFPStateMembership : (updated)
(iv) FFP : (updated)

1.7.1.3.1 StoreStatePair

First, the state pair is placed together in the blocks of all partitions; after, the states under consideration are put apart in only one partition, that means they are made adjacent states.
Finally the number of reached adjacencies for every state member is updated.

(a) OVERVIEW :

{ begin }

1.7.1.3.1.1 InsertStatePairInAllBlocksOfFFP;

1.7.1.3.1.2 DivorceStatePair;

1.7.1.3.1.3 UpdateStateAdjacencies;

{ end };

(b) HIGHLIGHTS :

* 1.7.1.3.1.1 InsertStatePairInAllBlocksOfFFP : by construction, there are only two possibilities for the pair of states: the first state is a state member and the second not; or otherwise. Hence, the state not member is placed in the blocks where the state member is already present.
Tool: internal procedure Join(StateW,StateZ,FFP) which inserts StateW in all the blocks of FFP including StateZ.

* 1.7.1.3.1.2 DivorceStatePair : the state pair in focus must be placed in separated blocks in only one partition (remember that one state is a state member and the other not). The best partition for this purpose is one of the partitions including the set of state members adjacent to the state member under consideration.
(Hint: the states in one set of adjacent states to a given state are never adjacent to themselves).
Tools:
(a) SelectFor(State,BestBlock,BestPartition): chooses the block within one partition including the set of adjacent states to State.
(b) Divorce(State,ChoosedBlock,PartialPartition): transfer State from ChoosedBlock to the other one within the PartialPartition.

* 1.7.1.3.1.3 UpdateStateAdjacencies : by construction, each new state member has perhaps other adjacencies than the ones already booked.

(c) INPUTS :

(i) ConditionalPair : state pair supposed to be inserted into the FFP.

(ii) ReachedAdjacencies: array containing the current number of reached adjacencies for every state.

(iii) SetOfAdjStates : array containing the set of current adjacent states to every state.

(iv) FFPStateMembership: set with current state members.

(v) FFP : (current status)

(d) OUTPUTS :

(i) ReachedAdjacencies : (updated)
(ii) SetOfAdjStates : (updated)
(iii) FFPStateMembership : (updated)
(iv) FFP : (updated)

1.7.1.3.2 LookForOverlapping

This procedure looks for a state member owning less reached adjacencies than the maximal allowed, for the purpose of introducing a new state in the FFP adjacent to it.

(a) OVERVIEW :

```
1.7.1.3.2 LookForOverlapping

          { begin }

            { repeat : }

              { get a state };

            { until state = member and
                    reached adjacencies less than maximal
                    allowed };

          { end };
```

(b) INPUTS :

(i) StateZ : the state to be introduced.

(ii) ReachedAdjacencies : array containing the current number of reached adjacencies for every state.
adjacent states to every state.

(iii) FFPStateMembership : set with current state members.

(c) OUTPUTS :

(i) ConditionalPair : pair of states supposed to be introduced in the FFP. (only one state is already a member).

1.7.1.4 DetectNeighbouring

It is meaningful to try to place conditional state pairs related to the current PriorityHeadAux only if the state pair PriorityHeadAux is self codewise adjacent.
This procedure checks if the states from PriorityHeadAux are adjacents.

(a) OVERVIEW :

```
1.7.1.4 DetectNeighbouring

        { begin }

          { count how many partitions have StateX and
            StateY in the same block };

          { verify adjacency condition };

        { end }
```

(b) INPUTS :

(i) StateX : state under consideration.
(ii) StateY : state under consideration.
(iii) FFP : (current status).

(c) OUTPUTS :

(i) AdjacentStates : boolean answer concerning adjacency status between StateX and StateY.

1.7.2 ShapeFinalFamilyOfPartitions

State assignment depends on state position within a partition (states in the first block are coded with "0", in the second block coded with "1"). In order to shape the "best" block positions, this procedure takes into consideration the number of entries of the states in the NextStateTable.

(a) OVERVIEW :

1.7.2 ShapeFinalFamilyOfPartitions

{ begin }

{ compute for each state the number of entries in the NextStateTable };

{ the block presenting the biggest sum of entries in the NextStateTable is choosed to be place in the first position within one partition };

{ end };

(b) INPUTS :

(i) FFP : final family of partitions.
{ current status }

(ii) NextStateTable : as entries to this table you have an input pattern identifier and a present state ; as output you have a next state (next state = -1 means that the current input pattern is not related to the current present state).

(c) OUTPUTS :

(i) FFP : final family of partitions.
{ final status }

1.7.3 RecordFinalFamilyOfPartitions

After having built and shaped a FFP, the results are registered in a file matching the simplest version of the minimiser input format definition (Appendix 1; aiming to save time, only the necessary information is given to the minimiser, without comments).

(a) OVERVIEW :

1.7.3 RecordFinalFamilyOfPartitions

```
{ begin }

  { assign codes to states considering position in
    the partitions };

  { register results in minimiser input file
    according to format definition };

{ end };
```

(b) INPUTS :

(i) FFP : { final status }.

(c) OUTPUTS :

(i) file matching minimiser input file format.

1.7.4 MinimizeMachineStructure

MINAS has no internal procedure allowing minimisation of encoded FSM's. In order to achieve this goal, we have decided to use the library program MOM (Multiple Output Minimiser), available in the TUE-Lib.
For the purpose of information exchange between the two programs, the mailbox concept is needed. A mailbox is an object (a file) that two (or more) programs use to get messages or to put messages in.
To read or to write using mailboxes, MINAS uses the "MBX system calls" described in detail in the mailbox chapter of the "Apollo Domain - Programming With System Calls for Interprocess Communication" manual.

(a) OVERVIEW :

```
1.7.4 MinimizeMachineStructure

      { begin }

        { get departure time; that means, register time
          instant before minimiser becomes activated };

        1.7.4.1 MBX_Client_Priori;

        1.7.4.2 MBX_Client_Posteriori;

        { get arrival time; i.e register time instant
          when minimiser becames inactivated };

      { end }
```

(b) HIGHLIGHTS :

* 1.7.4.1 MBX_Client_Priori : MINAS is one client of the mailbox server (MOM is the other one).
This procedure sends a message to the server instructing that the data file for the minimiser has been closed.

* 1.7.4.2 MBX_Client_Posteriori : This procedure asks to the server if the results from the minimiser are available. The server gives a positive answer only when the file with results from the miniser is closed.

(c) INPUTS :

(i) Step : FFP are generated sequentially. The server receives information about the current Step.

(d) OUTPUTS :

(i) MOM_sec : minimiser execution time for the current Step.

1.7.5 AnalyzeMinimizedStructure

The results from MOM (a minimised FSM) are available in a file (format definition given in Annexe1).
MINAS gets data from this file in order to decide which assignment will be the best.

(a) OVERVIEW :

1.7.5 AnalyzeMinimizedStructure

```
{ begin }

  { read information from file with minimised
    structure };

  { if current ProductTerms (i.e., the minimal cover
    cardinality) is less than latest reference
    then }

      { keep the FFP as the best until now };

{ end };
```

(b) INPUTS :

(i) ProductTerms : read from minimiser file.

(ii) MinProductTerms : latest value.

(iii) Step : index related to current FFP.

(c) OUTPUTS :

(i) MinProductTerms : updated value.

(ii) FFPKey : chosed "best" FFP for the time being.

1.8 RecordBestFinalFamilyOfPartitions

After having decided for the "best" final family of partitions (that is, the assignment resulting on the smallest number of product terms), MINAS registers the results in a file matching the minimiser input format definition (Annexe 1).

(a) OVERVIEW :

1.8 RecordBestFinalFamilyOfPartitions

```
{ begin }

  { assign codes to states considering position in
    the partitions };

  { open minimiser input file };

  { write some comments regarding the assignment };

  { register results in minimiser input file
    according to format definition };

  { close file };

{ end }
```

(b) INPUTS :

(i) FFPKey : best final family of partitions;
{ final status }

(c) OUTPUTS :

(i) file matching minimiser input file format.

1.9 RecordLoopTime

Same remarks as for GetReferenceTime (1.1).
This procedure computes and shows the algorithm's global execution time, including the required time for minimisation.

V. PROGRAM

```
program MINAS(input,output,PrimitiveData,Register,Mini);

  {##########################################################################}

                        { INCLUDE FILES }

  {##########################################################################}


  %INCLUDE '/sys/ins/base.ins.pas';
  %INCLUDE '/sys/ins/error.ins.pas';
  %INCLUDE '/sys/ins/mbx.ins.pas';
  %INCLUDE '/sys/ins/pgm.ins.pas';
  %INCLUDE '/sys/ins/time.ins.pas';
  %INCLUDE '/sys/ins/cal.ins.pas';


  {##########################################################################}

                { DATA STRUCTURE AND GLOBAL VARIABLES }

  {##########################################################################}


  const DataFile = 'FSM06.DAT';              { file with primitive data }
        RecordFile = 'FSM06.DEF';            { file with results }
        MomFile = 'FSM06.MIN';               { answer from MOM }

  const StateRange = 7;                      { S: Number of States }
        StatePairPositionRange = 21;         { (S -1)*S div 2 }
        PartitionRange = 3;                  { k; 2**(k-1) < S =< 2**k }

  const InputBitRange = 2;                   { Number of input lines }
        InputPatternRange = 5;               { IP: Number of different inputs}
        InputPairPositionRange = 10;         { (IP -1)*IP div 2 }

  const OutputBitRange = 2;                  { Number of output lines }
        OutputPatternRange = 4;              { Number of different outputs }

  const MNP = 2;                             { parameter, normaly = k/2 }
        MNSC = 7;                            { parameter, normaly = S or S+1 }
        Cost = 1.0;                          { parameter, range 0 .. 1 }


  {.........................................................................}


  type NonNegInteger = 0..maxint;
       Bits = 0..2;

  type States = 1..StateRange;
       SetOfStates = set of States;
       Block = SetOfStates;
       ProperPartition = record BlockA,BlockB: Block end;
       FamilyOfProperPartitions =array[1..PartitionRange] of ProperPartition;
```

```
type NextStates = -1..StateRange;

type StatePairSeries = 1..StatePairPositionRange;
     InputPairSeries = 1..InputPairPositionRange;
     InputSeries =  1..InputPatternRange;

     InputBitSeries = 1..InputBitRange;
     OutputSeries = 1..OutputPatternRange;
     OutputBitSeries = 1..OutputBitRange;
     PartitionSeries = 1..PartitionRange;
     MNSCSeries = 1..MNSC;

type BlockRange = 1..2;
     InputBitMatching = 0..InputBitRange;
     MaxField = 1..3;

type VectorStatePairPosition =array[StatePairSeries] of NonNegInteger;
     VectorCounter = array[StatePairSeries] of real;
     VectorInputDef = array[InputBitSeries] of Bits;
     VectorOutputDef = array[OutputBitSeries] of Bits;

type VectorCounterMNSC = array[MNSCSeries] of real;
     VectorStatePairPosMNSC = array[MNSCSeries] of NonNegInteger;

type StackPointer = ^StackComponent;
     StackComponent = record
                        Value:NonNegInteger;
                        Next: StackPointer
                      end;

type MatrixList = ^MatrixComponent;
     MatrixComponent = record
                        Vector: VectorInputDef;
                        Next: MatrixList
                       end;

type Crosses = record
                 Afinity: 0..2;
                 DubbleCross: InputBitMatching;
               end;


{..............................................................}


var PrimitiveData: text;    { file of NonNegInteger }
    Register,Mini: text;    { file of NonNegInteger }

var ConditionalTable: array[StatePairSeries] of StackPointer;
    ConditionalAdjacencies: array[StatePairSeries] of StackPointer;

var FFP,FFPKey: FamilyOfProperPartitions;
    FFPStateMembership: SetOfStates;
    NumberOfStateMembers: 0..StateRange;

var SetOfAdjStates: array[States] of SetOfStates;
    ReachedAdjacencies: array[States] of NonNegInteger;

var StateX,StateY: States;
```

```
var StatePairPosition: StatePairSeries;
    Location,PriorityHead: StatePairSeries;

var Item: NonNegInteger;

var VectorInputState: VectorCounter;
    VectorOutputState: VectorCounter;
    VectorDontCare: VectorCounter;
    VectorUnconditional: VectorCounter;
    VectorEstimationTotal: VectorCounter;
    VectorPriority: VectorStatePairPosition;

var NextStateTable: array[States,InputSeries] of NextStates;
    OutputTable: array[States,InputSeries] of OutputSeries;
    InputMatrixDef: array[InputSeries] of VectorInputDef;
    OutputMatrixDef: array[OutputSeries] of VectorOutputDef;

var VectorInputCorrelation: array[InputPairSeries] of  Crosses;
    VectorInputAutoCorrelation: array[InputSeries] of  Crosses;

var Bit: Bits;
    Option: 1..2;

var BestPartition : PartitionSeries;
    BestBlock     : BlockRange;
    Adjacencies   : NonNegInteger;
    SumAdjacencies: real;

var MinLogicGates,MinProductTerms,MinLocation: NonNegInteger;
    Step,MinStep: NonNegInteger;

var statrec: status_$t;     { error status after opening a file}
    clock: time_$clock_t;  { internal time }
    ref_sec,loop_sec: linteger;  { readable time in sec }
    MOM_sec,dep_sec,arr_sec: linteger; { MOM time }


{#########################################################################}

                                { TOOLS }

{#########################################################################}


procedure Push(X: NonNegInteger; var Stack: StackPointer);

  var NewComponent: StackPointer;

  begin
    new(NewComponent);
    with NewComponent^ do
      begin Value := X; Next := Stack end;
    Stack := NewComponent
  end; { Push }

{....................................}

procedure Pop(var X: NonNegInteger; var Stack: StackPointer);

  var OldComponent: StackPointer;
```

```
  begin
    OldComponent := Stack;
    with OldComponent^ do
      begin X := Value; Stack := Next end;
    dispose(OldComponent)
  end; { Pop }

{...................................}

procedure PushVector(X: VectorInputDef; var Matrix: MatrixList);

  var NewComponent: MatrixList;

    begin
      new(NewComponent);
      with NewComponent^ do
        begin Vector := X; Next := Matrix end;
      Matrix:= NewComponent
    end; { PushVec }

{...................................}

 procedure PopVector(var X: VectorInputDef; var Matrix: MatrixList);

   var OldComponent: MatrixList;

     begin
       OldComponent := Matrix;
       with OldComponent^ do
         begin X := Vector; Matrix := Next end;
       dispose(OldComponent)
     end; { PopVec }


{........................................................................}


procedure Order( var A,B: States);
  { this procedure examines and, if necessary, exchanges the values of    }
  { A and B so that the value of A is smaller than the value of B.        }

  var T: 1..StateRange;

  begin
    if A > B then begin T := A; A := B; B := T end
  end; { Order }

{...................................}

procedure Increment( var A: NonNegInteger);
  { the goal of this procedure is ... }

  begin
    A := A +1
  end; { increment }

{...................................}

procedure AddTo( var R: real; S: NonNegInteger);
  { the goal of this procedure is ... }
```

```
  begin
    R := R + S;
  end; { add }
{..................................}

function Power( A: NonNegInteger): NonNegInteger;
  { Computes 2 raised to the power A }

  var I,Answer: NonNegInteger;

  begin
    Answer :=1;
    for I := 1 to A do Answer := 2*Answer;
    Power := Answer;
  end; {Power}

{..................................}

function Field( A: NonNegInteger): MaxField;
  { returns field format for printing integers }

  begin
    if A < 10 then Field := 1
    else
      if A > 99 then Field := 3
      else Field := 2;
  end; {Field}


{..........................................................................}

procedure Locate(FirstState,SecondState: States;
                 var StatePairPosition : StatePairSeries);
  { this procedure receives a state pair (FirstState < SecondState) and    }
  { gives back its respective position in the ordered series of StatePairs}


  begin

    StatePairPosition := StateRange*(FirstState -1) +
                         SecondState - FirstState -
                         ((FirstState -1)*FirstState)div(2);

  end; { Locate }

{..................................}

procedure Decode(StatePairPosition: StatePairSeries;
                 var FirstState,SecondState: States);
  { this procedure gets a state pair position in the series of ordered     }
  { state pairs and computes the constitutive elements (First,SecondState)}

  var InversePosition {of  StatePair } : 1..StatePairPositionRange;
      Size {of the set of StatePairs with same FirstState} : 0..StateRange;

  begin

    InversePosition := StatePairPositionRange - StatePairPosition +1;
    Size := 0;
```

```
    FirstState  := StateRange - Size;
    SecondState := StatePairPosition +
                   (FirstState*(FirstState +1))div(2) -
                   StateRange*(FirstState -1);

  end; { Decode }

{..................................}

procedure DecodeInp(InputPairPosition: InputPairSeries;
                    var FirstPattern,SecondPattern: InputSeries);
  { this procedure gets a input pair position in the series of ordered     }
  { input pairs and computes the constitutive elements (First,SecPattern) }

  var InversePosition {of  InputPair in the series} : InputPairSeries;
      Size: 0..InputPatternRange;

  begin

    InversePosition := InputPairPositionRange - InputPairPosition +1;
    Size := 0;
    repeat Size := Size +1 until InversePosition <= (Size*(Size +1))div(2);

    FirstPattern  := InputPatternRange - Size;
    SecondPattern := InputPairPosition +
                     (FirstPattern*(FirstPattern +1))div(2) -
                     InputPatternRange*(FirstPattern -1);

  end; { DecodeInp }

{...................................................................}


procedure QuickSort( var Item : VectorCounter;
                     var Locus: VectorStatePairPosition);
  { this is the quick sort algorithm for sorting list Item and preserve    }
  { the previous correspondingly StatePairPosition in Locus.               }

  var Left,Right: StatePairSeries;

  procedure QS( var Item  : VectorCounter;
                var Locus : VectorStatePairPosition;
                Left,Right: StatePairSeries);
    { recursive algorithm }

    var I,J,L: NonNegInteger; var X,Y: real;

    begin

      I := Left; J := Right;
      X := Item[(Left+Right)div(2)];
      while I <= J do
        begin
          while (Item[I] > X)and(I < Right) do I := I +1;
          while (X > Item[J])and(J > Left ) do J := J -1;
          if I <= J then
            begin
              Y := Item[]; L := Locus[I];
              Item[I] := Item[J]; Locus[I] := Locus[J];
```

```
            I := I +1; J := J -1;
          end;
        end;
      if (Left < J)  then QS(Item,Locus,Left,J);
      if (I < Right) then QS(Item,Locus,I,Right);

    end; { QS }

  begin { protocol }

    Left :=1; Right := StatePairPositionRange;

    QS(Item,Locus,Left,Right);

  end; { QuickSort }

{..............................}


procedure ShortQuickSort( var Item : VectorCounterMNSC;
                          var Locus: VectorStatePairPosMNSC);

  var Left,Right: MNSCSeries;

  procedure QS( var Item  : VectorCounterMNSC;
                var Locus : VectorStatePairPosMNSC;
                Left,Right: MNSCSeries);
    { recursive algorithm }

    var I,J,L: NonNegInteger; var X,Y: real;

    begin

      I := Left; J := Right;
      X := Item[(Left+Right)div(2)];
      while I <= J do
        begin
          while (Item[I] > X)and(I < Right) do I := I +1;
          while (X > Item[J])and(J > Left ) do J := J -1;
          if I <= J then
            begin
              Y := Item[I]; L := Locus[I];
              Item[I] := Item[J]; Locus[I] := Locus[J];
              Item[J] := Y; Locus[J] := L;
              I := I +1; J := J -1;
            end;
        end;
      if (Left < J)  then QS(Item,Locus,Left,J);
      if (I < Right) then QS(Item,Locus,I,Right);

    end; { QS }

  begin { protocol }

    Left :=1; Right := MNSC;

    QS(Item,Locus,Left,Right);

  end; { ShortQuickSort }
```

```
{#######################################################################}

                              { PROCEDURES }

{#######################################################################}


procedure GetReferenceTime;

  {.................................}

  begin
    cal_$get_local_time(clock);
    ref_sec := cal_$clock_to_sec(clock);
    MOM_sec := 0;
  end;

{#######################################################################}


procedure GetPrimitiveData;

  {.................................}

  var VectorInput: VectorInputDef;
      VectorOutput: VectorOutputDef;

  var PresentState: States;
      NextState: NextStates;
      InputPattern,ExistingInputPatterns: 1..InputPatternRange;
      OutputPattern,ExistingOutputPatterns: 1..OutputPatternRange;

  var IBit: 1..InputBitRange; OBit: 1..OutputBitRange;

  var Equivalence: boolean;
      Item: NonNegInteger;

  {.................................}

  begin

    { initial conditions }
    for Item :=1 to OutputBitRange do
        OutputMatrixDef[1,Item] := 2;
    ExistingOutputPatterns := 1; { dont care label }
    for Item :=1 to InputBitRange do
        InputMatrixDef[1,Item] := 2;
    ExistingInputPatterns := 1;  { dont care label }
    for PresentState :=1 to StateRange do
     for Item :=1 to InputPatternRange do
      begin
        NextStateTable[PresentState,Item] := -1; { not allowed next_state}
        OutputTable[PresentState,Item] := 1;     { dont cares }
      end;

    { open file ...}

    open(PrimitiveData,DataFile,'OLD',statrec.all);
    if statrec.all=0 then reset(PrimitiveData)
    else writeln('There is no file named ',Datafile);
```

```
{ receive data ...}
while not eof(PrimitiveData) do
  begin

    for IBit :=1 to InputBitRange do
      begin read(PrimitiveData,Bit);VectorInput[IBit]:=Bit  end;
    read(PrimitiveData,PresentState);read(PrimitiveData,NextState);
    for OBit :=1 to OutputBitRange do
      begin read(PrimitiveData,Bit);VectorOutput[OBit]:=Bit end;
    readln(PrimitiveData);

    Item :=0;
    repeat
      Item := Item +1;
      Equivalence := true;
      for IBit :=1 to InputBitRange do
        begin
          Equivalence := (Equivalence)and
                      (VectorInput[Ibit] = InputMatrixDef[Item,IBit])
          end;
    until (Item = ExistingInputPatterns) or Equivalence ;
    if Equivalence then InputPattern := Item
    else
      begin
        ExistingInputPatterns := ExistingInputPatterns +1;
        InputPattern := ExistingInputPatterns;
        for IBit := 1 to InputBitRange do
          InputMatrixDef[InputPattern,IBit] := VectorInput[Ibit];
      end;

    Item :=0;
    repeat
      Item := Item +1;
      Equivalence := true;

      for OBit :=1 to OutputBitRange do
        begin
          Equivalence :=(Equivalence)and
                   (VectorOutput[OBit] = OutputMatrixDef[Item,OBit]);
        end;
    until (Item = ExistingOutputPatterns) or Equivalence ;
    if Equivalence then OutputPattern := Item
    else
      begin
        ExistingOutputPatterns := ExistingOutputPatterns +1;
        OutputPattern := ExistingOutputPatterns;
        for OBit := 1 to OutputBitRange do
            OutputMatrixDef[OutputPattern,OBit] := VectorOutput[Obit];
      end;

    NextStateTable[PresentState,InputPattern] := NextState;
    OutputTable[PresentState,InputPattern] := OutputPattern;

  end; { receive data }

 close(PrimitiveData);

end; { GetPrimitiveData }
```

```
{###########################################################################}

procedure ComputeInputStateAdjacencyConditions;

  {................................}


  var BookedSubspacesList,SubspacesWaitingList: MatrixList;

  var CoincMatrixA,CoincMatrixB:array[InputSeries] of VectorInputDef;
      Virginity: array[StatePairSeries] of boolean;

  var VectorPoint: VectorInputDef;

  var PresentState,StateA,StateB: States;
      NextStateA,NextStateB: NextStates;
      StatePairPosition: StatePairSeries;
      PatternA,PatternB: InputSeries ;
      InputPairPosition: InputPairSeries;

  var Mismatching,DubbleX: InputBitMatching;
      Item: NonNegInteger;


  {................................}


  procedure CompareBitByBitInput(PatternA,PatternB: InputSeries;
                                var Mismatching: InputBitMatching;
                                var DubbleX: InputBitMatching);

    var Item: 1..InputBitRange;
        BitA,BitB: Bits; { Bit = 2 ...> don't care! }

    begin

      Mismatching := 0; DubbleX := 0;

      for Item :=1 to InputBitRange do
        begin

          BitA := InputMatrixDef[PatternA,Item];
          BitB := InputMatrixDef[PatternB,Item];

          if (BitA <> BitB)and(not((BitA =2)or(BitB =2))) then
             Mismatching := Mismatching +1;
          if (BitA =2)and(BitB =2) then
             DubbleX := DubbleX +1;

        end;

    end; { Compare Input }

  {+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


  procedure FullInInputStateCoincidenceMatrix;

    var NextStateT: NextStates;
```

```
    InputPattern: InputSeries;

  { VectorPoint = vector input pattern definition  with don't care bits }

  begin

    for InputPattern :=1 to InputPatternRange do
      begin
        NextStateT := NextStateTable[PresentState,InputPattern];
        if (NextStateT = NextStateA) then
          begin
            CoincMatrixA[InputPattern] := InputMatrixDef[InputPattern];
            CoincMatrixB[InputPattern] := VectorPoint;
          end
        else
          if (NextStateT = NextStateB) then
            begin
              CoincMatrixB[InputPattern} := InputMatrixDef[InputPattern];
              CoincMatrixA[InputPattern] := VectorPoint;
            end
          else
            begin
              CoincMatrixA[InputPattern] := VectorPoint;
              CoincMatrixB[InputPattern] := VectorPoint;
            end;
      end;

  end; { full in CoincidenceMatrix }


{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


procedure GenerateAllVirtualSubspaces;
  { builts and stores subspaces }

  var SubspaceT: VectorInputDef;

  {...................................}

  procedure RecursiveCreation( IBit: InputBitSeries;
                               var SubspaceX: VectorInputDef );
    var IBitAux: InputBitSeries;
        BitA,BitB: Bits;

    procedure TestSequence;
      begin
        if IBitAux = InputBitRange then
          PushVector(SubspaceX,SubspacesWaitingList)
        else
          begin
            IBitAux := IBitAux +1;
            RecursiveCreation(IBitAux,SubspaceX);
          end
      end; { TestSequence }

    begin { recurtion }
    { the vectors that must be included in the subspace definition  }
    { are given by CoincMatrixA[PatternA] and CoincMatrixB[PatternB]}

      IBitAux := IBit;
```

```
      BitA := CoincMatrixA[PatternA][IBitAux];
      BitB := CoincMatrixB[PatternB][IBitAux];

      if (BitA = BitB) then
        begin SubspaceX[IBitAux] := BitA; TestSequence end
      else
        if (BitA <> 2)and(BitB <> 2) then
          begin SubspaceX[IBitAux] := 2; TestSequence end
        else
          if (BitA = 2) then
            begin
              SubspaceX[IBitAux] := BitB; TestSequence;
              SubspaceX[IBitAux] := 2; TestSequence;
            end
          else
            begin
              SubspaceX[IBitAux] := BitA; TestSequence;
              SubspaceX[IBitAux] := 2; TestSequence;
            end;
    end; { recursive algorithm }

  {.....................................}

  begin { generation }

    SubspacesWaitingList := nil;

    for PatternA := 1 to InputPatternRange do
      if CoincMatrixA[PatternA] <> VectorPoint then
        for PatternB := 1 to InputPatternRange do
          if CoincMatrixB[PatternB] <> VectorPoint then

            RecursiveCreation(1,SubspaceT); { initial conditions }

  end; { generate }


{+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


procedure KeepOnlyActualSubspaces;

  var EigenVectorsList: MatrixList;

  var SubspaceFocus: VectorInputDef;

  var Orthogonality: boolean;

  {.....................................}

  procedure FullInEigenVectorsSet;

    var NextStateT: NextStates;
        InputPattern: InputSeries;
        IBit: InputBitSeries;
        BitA,BitB: Bits;

    var EigenVectorAux: VectorInputDef;
        Correlation: boolean;

    begin
```

```
    EigenVectorsList := nil;

    for InputPattern :=1 to InputPatternRange do
      begin
        NextStateT := NextStateTable[PresentState,InputPattern];
        if (NextStateT = NextStateA)or
           (NextStateT = NextStateB)or
           (NextStateT = 0 {dontcare}) then
          begin
            Correlation := true;
            for IBit := 1 to InputBitRange do
              begin
                BitA := InputMatrixDef[InputPattern][IBit] ;
                BitB := SubspaceFocus[IBit];
                if BitA = BitB then
                  EigenVectorAux[IBit] := BitA
                else
                  if (BitA = 2) then
                    EigenVectorAux[IBit] := BitB
                  else
                    if (BitB = 2) then
                      EigenVectorAux[IBit] := BitA
                    else { BitA <> BitB }
                      Correlation := false;
              end;
            if Correlation then
              PushVector(EigenVectorAux,EigenVectorsList);
          end;
      end;

  end; { full in eigen_vectors }

{...................................}

procedure CheckBaseOrthogonality;

  var FutureEigenVectorsList:  MatrixList;
      EigenVectorsWaitingList: MatrixList;

  var EigenVectorA,EigenVectorB,EigenVectorAux: VectorInputDef;
      SubspaceT: VectorInputDef;

  var CancelationsIn,CancelationsOut,InternCancelations :NonNegInteger;
      Mismatching: InputBitMatching;
      Contraction : boolean;

  var BitA,BitB : Bits;
      IBit: InputBitSeries;

  begin

   FutureEigenvectorsList := nil; EigenVectorsWaitingList:= nil;
   CancelationsOut := 0;

   repeat

     CancelationsIn := CancelationsOut;InternCancelations := 0;

     while EigenVectorsList <> nil do
       begin
```

```
      PopVector(EigenVectorA,EigenVectorsList);Contraction := false;

      while (EigenVectorsList <> nil)and(not Contraction) do
        begin

          PopVector(EigenVectorB,EigenVectorsList);
          Mismatching := 0;
          for IBit :=1 to InputBitRange do
            begin
              BitA := EigenVectorA[IBit];BitB := EigenVectorB[IBit];
              if (BitA <> BitB) then Mismatching := Mismatching +1;
            end;

          if Mismatching =1 then { execute contraction }
            begin
              for IBit := 1 to InputBitRange do
                begin
                 BitA:= EigenVectorA[IBit];BitB:=EigenVectorB[IBit];
                 if BitA = BitB then EigenVectorAux[IBit] := BitA
                 else EigenVectorAux[IBit] := 2;
                end;
              Contraction := true;Increment(InternCancelations);
              PushVector(EigenVectorAux,FutureEigenVectorsList);
            end
          else
            begin
              EigenVectorAux := EigenVectorB;
              PushVector(EigenVectorAux,EigenVectorsWaitingList);
            end;

        end; { EigenVectorsList nil or contraction }

      if (not Contraction) then
        PushVector(EigenVectorA,FutureEigenVectorsList);

      while EigenvectorsWaitingList <> nil do
        begin
          PopVector(EigenVectorAux,EigenVectorsWaitingList);
          PushVector(EigenVectorAux,EigenVectorsList);
        end;

    end; { EigenVectorsList and WaitingList are empty }


  { full in main list again }

  while FutureEigenVectorsList <> nil do
    begin
      PopVector(EigenVectorAux,FutureEigenVectorsList);
      PushVector(EigenVectorAux,EigenVectorsList);
    end;


  CancelationsOut := CancelationsIn + InternCancelations;

until CancelationsIn = CancelationsOut;


{ EigenVectorsList has now subspace definitions ..................}
```

```
      Orthogonality := false;
      while (EigenVectorsList <> nil)and(not Orthogonality)do
        begin
          PopVector(SubspaceT,EigenVectorsList);
          if (SubspaceT = SubspaceFocus) then Orthogonality := true;
        end;


      { assure empty eigen_vectors list ................................}

      while EigenVectorsList <> nil do
        PopVector(SubspaceT,EigenVectorsList);

     end; { check orthogonality }

     {..............................}


   begin { check subspace existance }

     BookedSubspacesList := nil;

     while SubspacesWaitingList <> nil do
       begin
         PopVector(SubspaceFocus,SubspacesWaitingList);
         FullInEigenVectorsSet;
         CheckBaseOrthogonality;
         if Orthogonality then
           PushVector(SubspaceFocus,BookedSubspacesList);
       end;

   end; { keep only real subspaces }


{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


procedure DetectPossibleInputStateAdjacencies;

   var ExclusiveList: MatrixList;
       UpdatedExclusiveList,StandByList: MatrixList;
       WorkingListZ,WorkingListW: MatrixList;

   var SubspaceT: VectorInputDef;
       SubspaceA,SubspaceB: VectorInputDef;
       PossibleAdjacencies: InputBitMatching;

   var InclusionsIn,InclusionsOut,InternInclusions: NonNegInteger;
       Contraction : boolean;

   var BitA,BitB : Bits;
       DimensionA,DimensionB,Mismatching: InputBitMatching;
       Item: InputBitSeries;


   {.....................................}

   begin

     { full in working list for future information ....................}
```

```
WorkingListZ := nil; WorkingListW := nil;
while BookedSubspacesList <> nil do
  begin
    PopVector(SubspaceT,BookedSubspacesList);
    PushVector(SubspaceT,WorkingListW);
  end;
while WorkingListW <> nil do
  begin
    PopVector(SubspaceT,WorkingListW);
    PushVector(SubspaceT,BookedSubspacesList);
    PushVector(SubspaceT,WorkingListZ);
  end;

{ from booked subspaces keep only mutually exclusive ones..........}

while BookedSubspacesList <> nil do
  begin

    ExclusiveList := nil;
    PopVector(SubspaceA,BookedSubspacesList);
    PushVector(SubspaceA,ExclusiveList);

    { look for exclusive subspaces to SubspaceA ..................}

    while (WorkingListZ <> nil) do
      begin

        PopVector(SubspaceB,WorkingListZ);
        PushVector(SubspaceB,WorkingListW);

        Mismatching := 0;
        for Item :=1 to InputBitRange do
          begin
            BitA :=SubspaceA[Item];BitB := SubspaceB[Item];
            if (BitA <> BitB)and
               (not((BitA = 2)or(BitB = 2))) then
                 Mismatching := Mismatching +1;
          end;

        if (Mismatching <> 0) then { exclusive sets }
          PushVector(SubspaceB,ExclusiveList);

      end;{ WorkingListZ  is empty }


      { update exclusive list ....................................}

      StandByList := nil; UpdatedExclusiveList := nil;
      InclusionsOut := 0;

      repeat

        InclusionsIn := InclusionsOut; InternInclusions := 0;

        while ExclusiveList <> nil do
          begin

            PopVector(SubspaceA,ExclusiveList); Contraction := false;

            while (ExclusiveList <> nil)and(not Contraction) do
              begin
```

```
            PopVector(SubspaceB,ExclusiveList);
            Mismatching := 0; DimensionA := 0; DimensionB := 0;

            for Item := 1 to InputBitRange do
              begin
                BitA:=SubspaceA[Item];BitB:=SubspaceB[Item];
                if (BitA <> BitB)and
                   (not((BitA = 2)or(BitB = 2))) then
                     Mismatching := Mismatching +1;
                if BitA = 2 then DimensionA := DimensionA +1;
                if BitB = 2 then DimensionB := DimensionB +1;
              end;

            if Mismatching = 0 then { execute inclusion }
              begin
                if DimensionA >= DimensionB then
                  PushVector(SubspaceA,UpdatedExclusiveList)
                else PushVector(SubspaceB,UpdatedExclusiveList);
                Contraction := true;Increment(InternInclusions);
              end
            else { store information }
              PushVector(SubspaceB,StandByList);

          end; { exclusive list is empty or contraction }

          if not Contraction then
            PushVector(SubspaceA,UpdatedExclusiveList);

          while StandByList <> nil do
            begin
              PopVector(SubspaceT,StandByList);
              PushVector(SubspaceT,ExclusiveList);
            end;

       end; { exclusive list is empty }

    while UpdatedExclusiveList <> nil do
      begin
        PopVector(SubspaceT,UpdatedExclusiveList);
        PushVector(SubspaceT,ExclusiveList);
      end;

    InclusionsOut := InclusionsIn + InternInclusions;

  until InclusionsIn = InclusionsOut;


  { count  possible number of adjacencies .....................}

  PossibleAdjacencies := 0;
  while ExclusiveList <> nil do
    begin
      PopVector(SubspaceT,ExclusiveList);
      for Item := 1 to InputBitRange do
        if SubspaceT[Item] = 2 then
          PossibleAdjacencies := PossibleAdjacencies +1;
    end;


  { update max number of adjacencies ..........................}
```

```
          if PossibleAdjacencies > Adjacencies then
            Adjacencies := PossibleAdjacencies;


          { full in WorkingListZ again with all subspace definitions     }
          { aimind future information ..................................}

          while (WorkingListW <> nil) do
            begin
              PopVector(SubspaceT,WorkingListW);
              PushVector(SubspaceT,WorkingListZ);
            end;


      end; { booked subspaces list is empty ...........................}


    { assure empty WorkingListZ }

    while WorkingListZ <> nil do
      PopVector(SubspaceT,WorkingListZ);

  end; { Detect possible adjacencies }


{+++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


begin { protocol input_state adjacency conditions }

  for Item :=1 to StatePairPositionRange do VectorInputState[Item] :=0;
  for Item :=1 to InputBitRange do VectorPoint[Item] := 2; { only d.c. }

  { first check input binary correlation ..............................}
  writeln;
  write('computing input pattern correlation ..');

  for Item := 1 to InputPatternRange do
    begin

      PatternA := Item;
      CompareBitByBitInput(PatternA,PatternA,Mismatching,DubbleX);

      with VectorInputAutoCorrelation[PatternA] do
        begin Afinity := 0;DubbleCross := DubbleX end;
   end;

  for InputPairPosition :=1 to InputPairPositionRange do
    begin

     DecodeInp(InputPairPosition,PatternA,PatternB);
     CompareBitByBitInput(PatternA,PatternB,Mismatching,DubbleX);

      with VectorInputCorrelation[InputPairPosition] do
        begin
          if Mismatching = 0 then Afinity := 0
          else
            if Mismatching = 1 then Afinity := 1
            else Afinity := 2;
          DubbleCross := DubbleX;
```

```
        end;

      end;

     { now compute adjacency conditions ................................}

     writeln;

     for PresentState :=1 to StateRange do
       begin

         writeln;

         write('computing input-state adj cond.      ,  step:');
         write(PresentState:3,' /',StateRange:3);

         for Item := 1 to StatePairPositionRange do Virginity[Item] := true;

         for InputPairPosition :=1 to InputPairPositionRange do
           begin

             DecodeInp(InputPairPosition,PatternA,PatternB);
             NextStateA := NextStateTable[PresentState,PatternA];
             NextStateB := NextStateTable[PresentState,PatternB];

             if (NextStateA <> NextStateB)
                and(NextStateA > 0)and(NextStateB > 0) then
                begin
                  StateA:= NextStateA; StateB := NextStateB;
                  Order(StateA,StateB);
                  Locate(StateA,StateB,StatePairPosition);
                  if Virginity[StatePairPosition] then
                   begin
                     FullInInputStateCoincidenceMatrix;
                     GenerateAllVirtualSubspaces;
                     KeepOnlyActualSubspaces;
                     Adjacencies := 0;
                     DetectPossibleInputStateAdjacencies;
                     AddTo(VectorInputState[StatePairPosition],Adjacencies);
                     Virginity[StatePairPosition] := false;
                   end;
               end;
           end;

       end;

   end; { input-state adjacency conditions }


 {#####################################################################}


 procedure ComputeOutputStateAdjacencyConditions;

   {...............................}

   var StatePairPosition: StatePairSeries;
       InputPairPosition: InputPairSeries;
       InputPattern,InpPatternA,InpPatternB: InputSeries;
       StateA,StateB,StateP: States;
       NextStateA,NextStateB: NextStates;
```

```
    PatternA,PatternB: OutputSeries;

var Matching,XX: NonNegInteger;

{...............................}


procedure CountPossibleOutputStateAdjacencies;

  procedure CompareBitByBitOutput(PatternA,PatternB: OutputSeries;
                                  var Matching: NonNegInteger);
    var Item: 1..OutputBitRange;
        BitA,BitB: Bits; { Bit = 2 ...> don't care! }

    begin
      Matching := 0;
      for Item :=1 to OutputBitRange do

        begin
          BitA := OutputMatrixDef[PatternA,Item];
          BitB := OutputMatrixDef[PatternB,Item];
          if ((BitA = BitB) or (BitA = 2) or (BitB = 2)) then
             Matching := Matching +1;
        end;
    end; { Compare OutPut }


  begin

   if (NextStateA > 0) and (NextStateB > 0) then
     begin
       PatternA := OutputTable[StateA,InputPattern];
       PatternB := OutputTable[StateB,InputPattern];
       CompareBitByBitOutput(PatternA,PatternB,Matching);
       VectorOutputState[StatePairPosition] :=
         VectorOutputState[StatePairPosition] + Matching*Adjacencies;
     end;

   end; { count adj }

{...............................}


begin { protocol }

  writeln;writeln;
  write('computing output-state adj cond.     ,  step:');
  write(1:3,' /',StateRange:3);

  for Item :=1 to StatePairPositionRange do VectorOutputState[Item] :=0;

  StateP := StateRange;
  for StatePairPosition :=1 to StatePairPositionRange do
    begin

      Decode(StatePairPosition,StateA,StateB);

      if StateP <> StateA then
        begin
          StateP := StateA;
          writeln;
```

```
          write('computing output-state adj cond.     ,  step:');
          write(StateP +1:3,' /',StateRange:3);
        end;

      for InputPattern := 1 to InputPatternRange do
        begin

          XX := VectorInputAutoCorrelation[InputPattern].DubbleCross;
          Adjacencies := Power(XX);

          NextStateA :=NextStateTable[StateA,InputPattern];
          NextStateB :=NextStateTable[StateB,InputPattern];
          CountPossibleOutputStateAdjacencies;

        end;

      for InputPairPosition :=1 to InputPairPositionRange do
        if VectorInputCorrelation[InputPairPosition].Afinity = 0 then

          begin

            XX :=VectorInputCorrelation[InputPairPosition].DubbleCross;

            Adjacencies := Power(XX);

            DecodeInp(InputPairPosition,InpPatternA,InpPatternB);

            NextStateA := NextStateTable[StateA,InpPatternA];
            NextStateB := NextStateTable[StateB,InpPatternB];
            CountPossibleOutputStateAdjacencies;

            NextStateA := NextStateTable[StateA,InpPatternB];
            NextStateB := NextStateTable[StateB,InpPatternA];
            CountPossibleOutputStateAdjacencies;

          end;

    end;

  end; { output-state adjacency conditions }

{######################################################################}


procedure ComputePresentStateNextStateAdjacencyConditions;
  { this procedure also combines adjacency conditions and  gives an    }
  { estimation of the total number of adjacencies for each state pair }

  {..............................}

  var VectorLineAux: VectorCounter;
      VectorLocusAux: VectorStatePairPosition;

  var StatePairPosition: StatePairSeries;
      NextStatePairPosition: StatePairSeries;
      InputPairPosition: InputPairSeries;
      InputPattern,PatternA,PatternB: InputSeries;
      StateA,StateB,StateZ,StateW,StateP: States;
      NextStateA,NextStateB: NextStates;
```

```
var Item,XX: NonNegInteger;
    Counter: 0..StatePairPositionRange;
    Admission: boolean;


{.................................}

procedure CountPossiblePresStNextStAdjacencies;

  var StateZ,StateW: States;

  begin

    if ((NextStateA > 0)and(NextStateB >= 0))or
       ((NextStateA >= 0)and(NextStateB > 0)) then
       if (NextStateA = NextStateB)or
          (NextStateA = 0)or(NextStateB = 0) then
          AddTo(VectorDontCare[StatePairPosition],Adjacencies)
       else { NextStateA <> NextStateB <> 0 }
         begin
           StateZ:= NextStateA;StateW := NextStateB;
           Order(StateZ,StateW);
           Locate(StateZ,StateW,NextStatePairPosition);
           AddTo(VectorLineAux[NextStatePairPosition],Adjacencies);
         end;

  end; { count adj }
```

```
{..................................}

begin { protocol }

  { reset initial conditions for dont-cares }
  for Item :=1 to StatePairPositionRange do
    begin
      VectorDontCare[Item] :=0;
    end;

  { combine possible adjacencies for each state pair ..................}


  writeln;writeln;
  write('computing pres-next--state adj cond. ,  step:');
  write(1:3,' /',StateRange:3);

  StateP := StateRange;
  for StatePairPosition := 1 to StatePairPositionRange do

    begin

      { reset initial conditions }
      for Item := 1 to StatePairPositionRange do
        begin VectorLineAux[Item] := 0; VectorLocusAux[Item] := Item end;

      { compute present-state--next-state adjacency conditions }

      Decode(StatePairPosition,StateA,StateB);

      if StateP <> StateA then
        begin
          StateP := StateA;
          writeln;
          write('computing pres-next--state adj cond. ,  step:');
          write(StateP +1:3,' /',StateRange:3);
        end;


      for InputPattern :=1 to InputPatternRange do
        begin

          XX := VectorInputAutoCorrelation[InputPattern].DubbleCross;
          Adjacencies := Power(XX);

          NextStateA := NextStateTable[StateA,InputPattern];
          NextStateB := NextStateTable[StateB,InputPattern];
          CountPossiblePresStNextStAdjacencies;

        end;

      for InputPairPosition :=1 to InputPairPositionRange do
        if VectorInputCorrelation[InputPairPosition].Afinity = 0 then

          begin

            XX :=VectorInputCorrelation[InputPairPosition].DubbleCross;
            Adjacencies := Power(XX);
```

```
      DecodeInp(InputPairPosition,PatternA,PatternB);

      NextStateA := NextStateTable[StateA,PatternA];
      NextStateB := NextStateTable[StateB,PatternB];
      CountPossiblePresStNextStAdjacencies;

      NextStateA := NextStateTable[StateA,PatternB];
      NextStateB := NextStateTable[StateB,PatternA];
      CountPossiblePresStNextStAdjacencies;

    end;

{ compute independently reached combined adjacencies ............}

VectorUnconditional[StatePairPosition] :=
  PartitionRange*VectorDontCare[StatePairPosition] +
  (PartitionRange -1)*VectorLineAux[StatePairPosition] +
  (PartitionRange -1)*VectorInputState[StatePairPosition] +
  VectorOutputState[StatePairPosition];

{ compute combined adjacencies conditional to block placement , ..}
{ taking into account present-state--next-state and input-state ..}
{ dependencies ..............................................}

 for Item :=1 to StatePairPositionRange do
   if VectorLineAux[Item] > 0 then
     VectorLineAux[Item] := VectorLineAux[Item] +
                            VectorInputState[Item];


{ sort number of combined conditional adjacencies ...............}

VectorLineAux[StatePairPosition] := 0; {already updated }
QuickSort(VectorLineAux,VectorLocusAux);


{ store in ConditionalTable only the necessary priorities.........}

ConditionalTable[StatePairPosition] := nil;
ConditionalAdjacencies[StatePairPosition] := nil;
SumAdjacencies := 0;

for Counter := 1 to MNSC do

  begin

    Item := VectorLocusAux[Counter];
    Push(Item,ConditionalTable[StatePairPosition]);

    Item := trunc(VectorLineAux[Counter]);
    Push(Item,ConditionalAdjacencies[StatePairPosition]);
    SumAdjacencies := SumAdjacencies + Item;

  end;


 { estimation of the total number of adjacencies .........}

VectorEstimationTotal[StatePairPosition] :=
  VectorUnconditional[StatePairPosition] + MNP*SumAdjacencies;
```

```
      end;

     {..............................}

   end; { ComputeCombinedAdjacencies }


  {######################################################################}

  procedure  SortPriorities;
    { this procedure orders the state pair priorities }

    {..............................}

   var VectorAux: VectorCounter;
       VectorLocusAux: VectorStatePairPosition;

   var Counter: 1..StatePairPositionRange;

   begin

     writeln;writeln;writeln('shaping priorities ..');
     writeln;writeln('...');writeln;

     for Counter := 1 to StatePairPositionRange do
       begin
        VectorLocusAux[Counter] := Counter;
        VectorAux[Counter] := VectorEstimationTotal[Counter];
       end;

     QuickSort(VectorAux,VectorLocusAux);

     for Counter := 1 to StatePairPositionRange do
       VectorPriority[Counter] := VectorLocusAux[Counter];

   end; { sort priorities }


  {######################################################################}


  { procedures related to GenerateFinalFamiliesOfPartitionsBasedOnPriorities:}


  {+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


  procedure BuiltFinalFamilyOfPartitions;

    var ReferencePosition,ConditionalPair: StatePairSeries;
        PriorityHeadAux: StatePairSeries;
        StateA,StateB,StateZ,StateW: States;

    var Item: NonNegInteger;
        LocationAux,Counter: 0..StatePairPositionRange;
        Admission,AdjacentStates: boolean;

    var VectorLocus: array[1..MNSC] of NonNegInteger;
```

```
{:::::::::::::::::::::::::::::::::::::::::::::::::::}

procedure MakeLinkForStatePair;

  var ConditionalPairTemp: StatePairSeries;

  var StateA,StateB: States;

    var AdjX,AdjY: 0..PartitionRange;
        Partition: PartitionSeries;

  {..............................................}

  procedure LookForOverlapping;

    var Item: 0..StateRange;
        State: States;
        Admission: boolean;

    begin

      Admission := false;Item := 0;
      repeat
        Item :=Item +1; State := Item;
        if (State in FFPStateMembership)and
           (ReachedAdjacencies[State] < PartitionRange) then
           begin StateX := State; Admission := true end;
      until Admission;

      StateY := StateZ; Order(StateX,StateY);
      Locate(StateX,StateY,ConditionalPair);

    end; {look overlapping}

  {..............................................}

  procedure StoreStatePair;
    { ConditionalPair; StateX,StateY }

    var State: States;
        Partition: 1..PartitionRange;

    {.................................}

    procedure InsertStatePairInAllBlocksOfFFP;
    { the state pair (X,Y) is supposed to be inserted together }

      var Item: NonNegInteger;

      {...........................}

      procedure Join(StateW,StateZ : States;
                     var FFP: FamilyOfProperPartitions);
      { this procedure inserts StateW in  all the blocks including StateZ}

        var Item : 1..PartitionRange;

        begin
          for Item :=1 to PartitionRange do with FFP[Item] do
            begin
```

```
           if StateZ in BlockA then BlockA := BlockA + [StateW]
           else  BlockB := BlockB + [StateW];
          end;
      end; { Join }

      {.........................}

  begin { insertion state pair (X,Y) }

  { there are two possibilities : ..................................}

    if (StateX in FFPStateMembership)and
       (not(StateY in FFPStateMembership)) then
          Option :=1
    else {

      if (not(StateX in FFPStateMembership))and
         (StateY in FFPStateMembership)  then }
          Option :=2 ;


   { after choosing ............................................}

    case Option of

      1: { insert StateY in the partitions }
         begin
           Join(StateY,StateX,FFP);
           FFPStateMembership := FFPStateMembership +[StateY];
           NumberOfStateMembers := NumberOfStateMembers +1;
         end;

      2: { insert StateX in the partitions }
         begin
           Join(StateX,StateY,FFP);
           FFPStateMembership := FFPStateMembership +[StateX];
           NumberOfStateMembers := NumberOfStateMembers +1;
         end;

    end; { case }


    { update status ............................................}


    SetOfAdjStates[StateX] := SetOfAdjStates[StateX] + [StateY];
    SetOfAdjStates[StateY] := SetOfAdjStates[StateY] + [StateX];
    Increment(ReachedAdjacencies[StateX]);
    Increment(ReachedAdjacencies[StateY]);

  end; { insertion state pair }

{........................................}

procedure DivorceStatePair;
{ state pair (X,Y) must be apart in only one partition }

{...............................}

  procedure SelectFor(State: States;
                      var BestBlock: BlockRange;
```

```
                        var BestPartition : PartitionSeries);
  { The partition with the block including the SetOfAdjStates[State]}

   var Item: 1..PartitionRange;

   begin
     for Item := PartitionRange downto 1 do with FFP[Item] do
       begin

         if (SetOfAdjStates[State] <= BlockA) then
           begin BestPartition := Item; BestBlock := 1 end
         else
           begin
             if SetOfAdjStates[State] <= BlockB then
               begin BestPartition := Item; BestBlock := 2 end
             else; { try another partition }
           end;
       end;

   end; { Select }

{...................................}

  procedure Divorce(State: States;
                    ChoosedBlock : BlockRange;
                    var PartialPartition : ProperPartition);
   begin
     with PartialPartition do
       begin
         if ChoosedBlock = 1 then
           begin
             BlockA := BlockA - [State]; BlockB := BlockB + [State];
           end
         else
           begin
             BlockB := BlockB - [State]; BlockA := BlockA + [State];
           end;
       end;
   end; { Divorce }

{...................................}

  begin { divorce protocol }

   case Option of

     1: begin  { StateX is supposed to stay untouched }
          SelectFor(StateX,BestBlock,BestPartition);
          Divorce(StateY,BestBlock,FFP[BestPartition]);
        end;

     2: begin  { StateY is supposed to stay untouched }
          SelectFor(StateY,BestBlock,BestPartition);
          Divorce(StateX,BestBlock,FFP[BestPartition]);
        end;

   end; { case }

  { StateY is divorced from StateX once}

  end; { divorce procedure }
```

```
{.................................}

  procedure UpdateStateAdjacencies;
  { by construction each new state member has other adjacencies than }
  { the ones already booked ..                                      }

    var StateZ,StateW: States;
        StateA,StateB: States;
        Item,Matching: NonNegInteger;

    begin

      for StateZ :=1 to StateRange do
        if StateZ in FFPStateMembership then

          for StateW :=1 to StateRange do
            if StateW in FFPStateMembership then
              if not(StateW in SetOfAdjStates[StateZ]) then

                begin
                  Matching := 0;
                  StateA := StateZ; StateB := StateW;

                  Order(StateA,StateB);
                  for Item :=1 to PartitionRange do with FFP[Item] do
                    if ([StateA,StateB] <= BlockA) or
                       ([StateA,StateB] <= BlockB) then
                          Matching := Matching +1;
                  if Matching = (PartitionRange -1) then
                    begin
                      SetOfAdjStates[StateZ] :=
                          SetOfAdjStates[StateZ] + [StateW];
                      Increment(ReachedAdjacencies[StateZ]);
                    end;
                end;

    end; { UpdateAdjacencies }

  {.................................}

  begin { protocol store }

    { state pair under consideration is StateX, StateY }

    InsertStatePairInAllBlocksOfFFP;
    DivorceStatePair;
    UpdateStateAdjacencies;

  end;  { store state pair }

{..........................................}


begin { protocol MakeLink }

  Decode(ConditionalPair,StateX,StateY);
  AdjX := ReachedAdjacencies[StateX];
  AdjY := ReachedAdjacencies[StateY];
```

```
if ((AdjX < PartitionRange)and(AdjX <> 0)and(AdjY = 0))or
   ((AdjY < PartitionRange)and(AdjY <> 0)and(AdjX = 0)) then

   begin StoreStatePair end

else

  if ((AdjX = 0)and(AdjY = 0)) then

    begin

      StateA := StateX; StateB := StateY;
      ConditionalPairTemp := ConditionalPair;

      StateZ := StateA;
      LookForOverlapping; StorestatePair;


      if ReachedAdjacencies[StateA] < PartitionRange then
        begin
          StateX := StateA; StateY := StateB;
          ConditionalPair := ConditionalPairTemp;
          StoreStatePair;
        end

      else
        begin

          StateZ := StateB;
          LookForOverlapping; StoreStatePair;
        end;

    end

  else

    if ((AdjX = PartitionRange)and(AdjY = 0))or
       ((AdjY = PartitionRange)and(AdjX = 0)) then

       begin

         StateA := StateX; StateB := StateY;
         if AdjX = PartitionRange then StateZ := StateB
         else StateZ := StateA;
         LookForOverLapping;StoreStatePair;

       end

    else

      if (AdjX <> 0)and(AdjY <> 0) then  { do nothing }
      else
        begin
          writeln(' logic error ...');
          writeln(' AdjX :',AdjX,'   AdjY: ',AdjY);
          while true do;
        end;

end;{ make link }
```

```
{::::::::::::::::::::::::::::::::::::::::::::::::::}

procedure DetectNeighbouring;
  { Are StateX and StateY adjacent ? }

  var Matching,Partition: 0..PartitionRange;

  begin

    AdjacentStates := false;

    Matching := 0;
    for Partition :=1 to PartitionRange do with FFP[Partition] do
      if ([StateX,StateY] <= BlockA)or([StateX,StateY] <= BlockB) then
         Matching := Matching +1;

    if Matching = (PartitionRange -1) then AdjacentStates := true;

  end; { detect }


{::::::::::::::::::::::::::::::::::::::::::::::::::}


procedure FullInVectorAuxiliarWithConditionalPairs;
  { following ReferencePosition }

  {...............................}

  var WaitingListAdj,WaitingListPos: StackPointer;

  var ItemAdj,ItemPos: NonNegInteger;

      Sequencer: NonNegInteger;
      ReferenceWeight: NonNegInteger;
      MaxWeight, Weight: real;

  var VectorCondAdjAux: VectorCounterMNSC;
      VectorLocusAux: VectorStatePairPosMNSC;

  var Counter: 1..MNSC;
      ConditionalPair: 1..StatePairPositionRange;

  {...............................}


  begin  { protocol }

    { sort priorities ............................................}

    for Counter := 1 to MNSC do
      begin
        VectorLocusAux[Counter] := Counter;
        VectorCondAdjAux[Counter] := 0;
      end;

    WaitingListAdj := nil; WaitingListPos := nil;
```

```
    while ConditionalTable[ReferencePosition] <> nil do
      begin
        Pop(ItemAdj,ConditionalAdjacencies[ReferencePosition]);
        Pop(ItemPos,ConditionalTable[ReferencePosition]);
        Push(ItemAdj,WaitingListAdj); Push(ItemPos,WaitingListPos);
      end;

    Sequencer := 0;
    repeat
      Sequencer := Sequencer +1;
      Counter := Sequencer;
      Pop(ItemAdj,WaitingListAdj); Pop(ItemPos,WaitingListPos);
      Push(ItemAdj,ConditionalAdjacencies[ReferencePosition]);
      Push(ItemPos,ConditionalTable[ReferencePosition]);
      ConditionalPair := ItemPos;
      VectorLocusAux[Counter] := ConditionalPair;
      VectorCondAdjAux[Counter] :=  (PartitionRange -1)*ItemAdj +
        VectorUnconditional[ConditionalPair]
    until WaitingListPos = nil;

    ShortQuickSort(VectorCondAdjAux,VectorLocusAux);


    { choose reference weight ........................................}

    MaxWeight := VectorCondAdjAux[1];{ cond pair with greatest cond adj }

    Weight := Cost*MaxWeight; ReferenceWeight := trunc(Weight);


    { full in VectorAux ..............................................}

    for Counter:= 1 to MNSC do VectorLocus[Counter] :=0;

    for Counter := 1 to MNSC do
      begin
        ItemAdj := trunc(VectorCondAdjAux[Counter]);
        if ItemAdj > ReferenceWeight then
          VectorLocus[Counter] := VectorLocusAux[Counter]
      end;

  end; { full in vector aux }


{:::::::::::::::::::::::::::::::::::::::::::::::::::}


procedure  PlaceConditionalPairsFromVectorAuxiliar;

  var  Counter: 0..MNSC;
       EndCycle: boolean;

  begin

    Counter := 0; EndCycle := false;
    repeat
      Counter := Counter +1;
      if VectorLocus[Counter] = 0 then EndCycle := true
      else
        begin
          ConditionalPair := VectorLocus[Counter];
```

```
          MakeLinkForStatePair;
        end;
    until EndCycle or (Counter = MNSC) or
          (NumberOfStateMembers = StateRange);

  end; { place cond pairs }

{.................................}


begin { protocol }

  { PriorityHead := VectorPriority[Location] }


  { initial conditions}

     FFPStateMembership := []; NumberOfStateMembers :=0;
     for Item :=1 to PartitionRange do with FFP[Item] do
       begin BlockA := []; BlockB := [] end;
     for Item := 1 to StateRange do
       begin
         SetOfAdjStates[Item] := [Item];
         ReachedAdjacencies[Item] := 0;
       end;

  { place PriorityHead }

     Decode(PriorityHead,StateX,StateY);

     with FFP[1] do
       begin BlockA := [StateX]; BlockB := [StateY] end;
     for Item :=2 to PartitionRange do with FFP[Item] do
       begin BlockA :=[StateX]; BlockA := BlockA +[StateY] end;

     FFPStateMembership := FFPStateMembership + [StateX];
     FFPStateMembership := FFPStateMembership + [StateY];
     NumberOfStateMembers :=2;
     SetOfAdjStates[StateX] := SetOfAdjStates[StateX]+[StateY];
     SetOfAdjStates[StateY] := SetOfAdjStates[StateY]+[StateX];
     ReachedAdjacencies[StateX] :=1; ReachedAdjacencies[StateY] :=1;


  { place ConditionalPairs related to PriorityHead }

     ReferencePosition := PriorityHead;
     FullInVectorAuxiliarWithConditionalPairs;
     PlaceConditionalPairsFromVectorAuxiliar;


  { StoreOtherPairsFollowingPriorities }

    if NumberOfStateMembers <> StateRange then
      begin
        LocationAux := 0;
        repeat
          LocationAux := LocationAux +1;
          PriorityHeadAux := VectorPriority[LocationAux];
          if PriorityHeadAux <> PriorityHead then
            begin
              ConditionalPair := PriorityHeadAux;
```

```
                MakeLinkForStatePair;
                if NumberOfStateMembers <> StateRange then
                  begin
                    Decode(PriorityHeadAux,StateX,StateY);
                    DetectNeighbouring;
                    if AdjacentStates then
                      begin
                        ReferencePosition := PriorityHeadAux;
                        FullInVectorAuxiliarWithConditionalPairs;
                        PlaceConditionalPairsFromVectorAuxiliar;
                      end;
                  end;
              end;
          until NumberOfStateMembers = StateRange;
        end;

  end; { Built FFP }

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


procedure ShapeFinalFamilyOfPartitions;

  var BlockT: Block;
      State: States; NextState: NextStates;
      InputPattern: InputSeries;
      Partition: 1..PartitionRange;

  var SumEntriesA,SumEntriesB: NonNegInteger;
      StateZ: States;

  var Entry: array[States] of NonNegInteger;

  {..............................}


  begin { protocol }


      { compute state entries in next-state table ......................}

        { initial conditions }
        for State := 1 to StateRange do Entry[State] := 0;

        { identification loop }

        for State := 1 to StateRange do
          for InputPattern := 1 to InputPatternRange do
           begin
             NextState := NextStateTable[State,InputPattern];
             if NextState > 0 then
               begin StateZ := NextState; Increment(Entry[StateZ]) end;
           end;


      { shape partition blocks ......................................}


        for Partition := 1 to PartitionRange do  with FFP[Partition] do
```

```
        begin

          SumEntriesA := 0;
          for State := 1 to StateRange do if State in BlockA then
            SumEntriesA := SumEntriesA + Entry[State];
          SumEntriesB := 0;
          for State := 1 to StateRange do if State in BlockB then
            SumEntriesB := SumEntriesB + Entry[State];

          if SumEntriesB > SumEntriesA then
            begin
              BlockT := BlockA; BlockA := BlockB; BlockB := BlockT;
            end;

        end;

    end; { shape }

 {++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


 procedure RecordFinalFamilyOfPartitions;

   var State,PresentState: States;
       NextState: NextStates;
       Partition: 1..PartitionRange;
       IBit: 1..InputBitRange;
       OBit: 1..OutputBitRange;
       Item: NonNegInteger;

   var DefinitionMatrix: array[States,PartitionSeries] of boolean;

   var Key: char;

   begin

      { assign code to states ..........................................}

      for State := 1 to StateRange do   .
        begin
          for Partition :=1 to PartitionRange do with FFP[Partition] do
            if State in BlockA then
                DefinitionMatrix[State,Partition] := false
            else
                DefinitionMatrix[State,Partition] := true;
        end;


      { record results in file ........................................}

      open(Register,RecordFile,'UNKNOWN',statrec.all);

      if statrec.all = status_$ok then rewrite(Register)
      else ERROR_$PRINT(statrec);
      writeln(Register);

      reset(PrimitiveData);

      while not eof(PrimitiveData) do

        begin
```

```
        for Ibit :=1 to InputBitRange do
          begin
            read(PrimitiveData,Bit);
            case Bit of
              0: write(Register,'0 ');
              1: write(Register,'1 ');
              2: write(Register,'X ');
            end;
          end;

        read(PrimitiveData,PresentState);
        for Partition :=1 to PartitionRange do
          begin
            case DefinitionMatrix[PresentState,Partition] of
              false: write(Register,'0 ');
              true : write(Register,'1 ');
            end;
          end;

        write(Register,'| ');

        read(PrimitiveData,NextState);
        if NextState > 0 then
          for Partition :=1 to PartitionRange do
            begin
              case DefinitionMatrix[NextState,Partition] of
                false: write(Register,'. ');
                true : write(Register,'A ');
              end;
            end
         else
           for Partition := 1 to PartitionRange do
             write(Register,'- ');

        for OBit :=1 to OutputBitRange do
          begin
            read(PrimitiveData,Bit);
            case Bit of
              0: write(Register,'. ');
              1: write(Register,'A ');
              2: write(Register,'- ');
            end;
          end;

        writeln(Register); readln(PrimitiveData);

      end; { eof data }

    close(Register);

    { pass message }

    writeln;
    writeln('After MOM the process should continue .. ');

  end; { Record }

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
```

```
procedure MinimizeMachineStructure;

  {................................}

  procedure MBX_Client_Priori( Step : integer);

    label done;

   const mbx_name    = 'POBOX';
         mbx_namelen = sizeof(mbx_name);
         buf_len     = mbx_$msg_max;
         msg_buf_len = mbx_$msg_max;

   type  msg_t = array[1..mbx_$msg_max] of char;
         msg_ptr_t = ^msg_t;

   var  mbx_handle : univ_ptr;
        status     : status_$t;
        data_buf   : msg_t;
        msg_buf    : msg_t;
        msg_retptr : msg_ptr_t;
        msg_retlen : integer32;

   var  Letter3,Letter4,Letter5 : char;
        Digit3,Digit4,Digit5: integer;

   procedure CheckStatus;
     begin
       if( status.all <> status_$ok)and
         ( status.all <> mbx_$partial_record)  then
         begin error_$print(status);pgm_$exit end;
     end;


   begin

     mbx_$open( mbx_name,
                mbx_namelen,
                nil,
                0,
                mbx_handle,
                status);

     CheckStatus;

     { send message and  step  }

     Digit3 := (Step)div(100);
     Digit4 := ( Step - Digit3*100)div(10) ;
     Digit5 := Step - Digit3*100 - Digit4*10;

     Letter3 := chr(Digit3 + 48);
     Letter4 := chr(Digit4 + 48);    { ascii ! }
     Letter5 := chr(Digit5 + 48);

     data_buf[1] := 'A'; { MINAS sign }
     data_buf[2] := '!'; { exec. finished }
     data_buf[3] := Letter3;
     data_buf[4] := Letter4;
     data_buf[5] := Letter5;
```

```
   mbx_$put_rec( mbx_handle,
                 addr(data_buf),
                 buf_len,
                 status);

   mbx_$get_rec( mbx_handle,
                 addr(msg_buf),
                 msg_buf_len,
                 msg_retptr,
                 msg_retlen,
                 status);

     CheckStatus;

   done:

     mbx_$close( mbx_handle, status);
     CheckStatus;

 end; { priori communication }

 {..................................}

 procedure MBX_Client_Posteriori( Step : integer);

   label done;

const mbx_name    = 'POBOX';
      mbx_namelen = sizeof(mbx_name);
      buf_len     = mbx_$msg_max;
      msg_buf_len = mbx_$msg_max;

type  msg_t = array[1..mbx_$msg_max] of char;
      msg_ptr_t = ^msg_t;

var  mbx_handle : univ_ptr;
     status     : status_$t;
     data_buf   : msg_t;
     msg_buf    : msg_t;
     msg_retptr : msg_ptr_t;
     msg_retlen : integer32;

var msg_array: array[1..5] of char;
    i: integer;
    Letter3,Letter4,Letter5 : char;
    Digit3,Digit4,Digit5: integer;

procedure CheckStatus;
   begin
     if( status.all <> status_$ok)and
       ( status.all <> mbx_$partial_record)  then
       begin error_$print(status);pgm_$exit end;
   end;

begin { protocol }

  mbx_$open( mbx_name,
             mbx_namelen,
             nil,
             0,
             mbx_handle,
```

```
              status);

      CheckStatus;

      { wait loop until good answer }

      Digit3 := (Step)div(100);
      Digit4 := ( Step - Digit3*100)div(10) ;
      Digit5 := Step - Digit3*100 - Digit4*10;

      Letter3 := chr(Digit3 + 48);
      Letter4 := chr(Digit4 + 48);
      Letter5 := chr(Digit5 + 48);

      repeat

        data_buf[1] := 'A'; { MINAS sign }
        data_buf[2] := '?'; { request }
        data_buf[3] := Letter3;
        data_buf[4] := Letter4;
        data_buf[5] := Letter5;

        mbx_$put_rec( mbx_handle,
                      addr(data_buf),
                      buf_len,
                      status);

         mbx_$get_rec( mbx_handle,
                       addr(msg_buf),
                       msg_buf_len,
                       msg_retptr,
                       msg_retlen,
                       status);

         CheckStatus;

         for i := 1 to 5 do
           msg_array[i] := msg_retptr^[i];

       until msg_array[1] = 'G';

     done:

       mbx_$close( mbx_handle, status);
       CheckStatus;

     end; { posteriori communication }

    {.....................................}


    begin { protocol }

      { go to the mailbox }

       cal_$get_local_time(clock);           { get departure time }
       dep_sec := cal_$clock_to_sec(clock);

       MBX_Client_Priori(Step);
```

```
    { wait until MOM is finished }

     MBX_Client_Posteriori(Step);

     cal_$get_local_time(clock);          { get arrival time }
     arr_sec := cal_$clock_to_sec(clock);
     MOM_sec := MOM_sec + ( arr_sec - dep_sec);

  end; { minimization }

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}


procedure AnalyzeMinimizedStructure;

var ProductTerms: NonNegInteger;
    DontCares: NonNegInteger;
    Actives: NonNegInteger;
    LogicGates: NonNegInteger;


 begin

   { get data from MOM }

   open(Mini,MomFile,'OLD',statrec.all);
   if statrec.all = 0 then reset(Mini)
   else writeln('Difficulty opening ',MomFile);

   repeat readln(Mini);get(Mini) until Mini^ = 'M';

   repeat get(Mini) until ( Mini^ in ['1'..'9'] ); read(Mini,ProductTerms);

   repeat get(Mini) until ( Mini^ in ['1'..'9'] ); read(Mini,DontCares);

   repeat get(Mini) until ( Mini^ in ['1'..'9'] );read(Mini,Actives);

   close(Mini);


   { test if minimal solutuion }


   LogicGates := (InputBitRange + PartitionRange)*ProductTerms
                 - DontCares + Actives;

   writeln;writeln('State Assignment: ',Location:Field(Location),' :');
   write('ProductTerms : ',ProductTerms:Field(ProductTerms));
   writeln('  LogicGates : ',LogicGates:Field(LogicGates));
   writeln;write('...');
   writeln;writeln;writeln;

   if ProductTerms < MinProductTerms then
     begin
        MinProductTerms := ProductTerms;
        MinStep  := Step;
        FFPKey := FFP;
     end;

 end; { analyze }
```

```
{+++++++++++++++++++++++++++++++++++++++++++++++}

procedure  GenerateFinalFamiliesOfPartitionsBasedOnPriorities;

  begin

    MinProductTerms := maxint;

    Step := 0;

    repeat

      Step := Step + 1; Location := Step;

      PriorityHead := VectorPriority[Location];

      BuiltFinalFamilyOfPartitions;

      ShapeFinalFamilyOfPartitions;

      RecordFinalFamilyOfPartitions;

      MinimizeMachineStructure;

      AnalyzeMinimizedStructure;

    until (Location - MinLocation = 5)or
          (Step = StatePairPositionRange);

  end; { generate }

{####################################################################}

procedure RecordBestFinalFamilyOfPartitions;

  var State,PresentState: States;
      NextState: NextStates;
      Partition: 1..PartitionRange;
      IBit: 1..InputBitRange;
      OBit: 1..OutputBitRange;
      Item: NonNegInteger;

  var DefinitionMatrix: array[States,PartitionSeries] of boolean;

  var Key: char;

  begin

    writeln;writeln;
    writeln('Best State Assignment : '); writeln;
    FFP := FFPKey; Location := MinStep ; Step := MinStep ;
    PriorityHead := VectorPriority[Location];


    writeln('Final Family of Partitions for step',Step:3);
    writeln('estimated adjacencies :
```

```
         ',VectorEstimationTotal[PriorityHead]:1:0);
writeln;


  for Partition :=1 to PartitionRange do with FFP[Partition] do
    begin
      write('    { [ ');
      for State :=1 to StateRange do if State in BlockA then
        write(State:Field(State),' ');
      write('] ; [ ');
      for State :=1 to StateRange do if State in BlockB then
        write(State:Field(State),' ');
      write('] }');writeln;
    end;

  for State := 1 to StateRange do
    begin
      for Partition :=1 to PartitionRange do with FFP[Partition] do
        if State in BlockA then
            DefinitionMatrix[State,Partition] := false
        else
            DefinitionMatrix[State,Partition] := true;
    end;


  { record results in file ......................................}

  open(Register,RecordFile,'UNKNOWN',statrec.all);
  if statrec.all = status_$ok then rewrite(Register)
  else ERROR_$PRINT(statrec);


  writeln(Register);writeln(Register);
  writeln(Register,'"State Assignment : ',Location:Field(Location));

  for State :=1 to StateRange do
    begin
      write(Register,'"State',State:Field(State),' : ');
      for Partition :=1 to PartitionRange do
        begin
          case DefinitionMatrix[State,Partition] of
            false: write(Register,'0 ');
            true : write(Register,'1 ');
          end;
        end;
      writeln(Register);
   end;

  writeln(Register,'" ');
  write(Register,'$IN I');
  for IBit := 1 to InputBitRange -1 do
    write(Register,IBit:Field(IBit),',I');
  write(Register,InputBitRange:Field(InputBitRange));
  for Item := PartitionRange -1 downto 0 do
    write(Register,',PS',Item:Field(Item));
  writeln(Register);
  write(Register,'$OUT NS');
  for Item := PartitionRange -1 downto 1 do
    write(Register,Item:Field(Item),',NS');
  write(Register,'0');
  for OBit :=1 to OutputBitRange do
```

```
  write(Register,',O',OBit:Field(OBit));
writeln(Register);writeln(Register,'" ');

reset(PrimitiveData);

while not eof(PrimitiveData) do

  begin

    for Ibit :=1 to InputBitRange do
      begin
        read(PrimitiveData,Bit);
        case Bit of
          0: write(Register,'0 ');
          1: write(Register,'1 ');
          2: write(Register,'X ');
        end;
      end;

    read(PrimitiveData,PresentState);
    for Partition :=1 to PartitionRange do
      begin
        case DefinitionMatrix[PresentState,Partition] of
          false: write(Register,'0 ');
          true : write(Register,'1 ');
        end;
      end;

    write(Register,'| ');

    read(PrimitiveData,NextState);
    if NextState > 0 then
      for Partition :=1 to PartitionRange do
        begin
          case DefinitionMatrix[NextState,Partition] of
            false: write(Register,'. ');
            true : write(Register,'A ');
          end;
        end
     else
       for Partition := 1 to PartitionRange do
         write(Register,'- ');

    for OBit :=1 to OutputBitRange do
      begin
        read(PrimitiveData,Bit);
        case Bit of
          0: write(Register,'. ');
          1: write(Register,'A ');
          2: write(Register,'- ');
        end;
      end;

    writeln(Register); readln(PrimitiveData);

  end; { eof data }

close(Register);


{ echo state assignment }
```

```
      writeln;writeln('State Assignment: ',Step:Field(Step));
      write('MinProductTerms : ',MinProductTerms:Field(MinProductTerms));
      writeln('  LogicGates : ',MinLogicGates:Field(MinLogicGates));


      writeln;
      for State := 1 to StateRange do
        begin
          write('State',State:Field(State),' : ');
          for Partition :=1 to PartitionRange do with FFP[Partition] do
            if State in BlockA then write('0 ') else write('1 ');
          writeln;
        end;

      writeln;

  end;  { record best FFP }


 {#########################################################################}

 procedure RecordLoopTime;

   var abs_sec, abs_min : integer;
       hour,min,sec : integer;

   begin

     cal_$get_local_time(clock);
     loop_sec := cal_$clock_to_sec(clock);

     writeln('Execution time  :');
     write('  - global      :');
     abs_sec := loop_sec - ref_sec;
     abs_min := (abs_sec)div(60);
     sec := (abs_sec)mod(60);
     min := (abs_min)mod(60);
     hour:= (abs_min)div(60);
     writeln(hour:3,' h :',min:3,' m :',sec:3,' s.');

     write('  - without MOM :');
     abs_sec := loop_sec - ref_sec - MOM_sec;
     abs_min := (abs_sec)div(60);
     sec := (abs_sec)mod(60);
     min := (abs_min)mod(60);
     hour:= (abs_min)div(60);
     writeln(hour:3,' h :',min:3,' m :',sec:3,' s.');

    end; {record loop time }
```

```
{##########################################################################}
                        { MAIN PROGRAM }
{##########################################################################}

begin
  GetReferenceTime;
  GetPrimitiveData;
  ComputeInputStateAdjacencyConditions;
  ComputeOutputStateAdjacencyConditions;
  ComputePresentStateNextStateAdjacencyConditions;
  SortPriorities;
  GenerateFinalFamiliesOfPartitionsBasedOnPriorities;
  RecordBestFinalFamilyOfPartitions;
  RecordLoopTime;
end
{##########################################################################}.
```

VI. RESULTS

MINAS has been tested on a set of industrial finite state machines (Table VI.1). The state tables for these machines are available at Annexe 2.

The same set has been used to test the program KISS at the University of California, Berkeley, as reported in [2].
KISS is an approach for state assignment of finite state machines, based on symbolic minimisation of the FSM combinational component and on a related constrained encoding problem.

The results obtained by MINAS compare favourably to KISS, with regard to the PLA area, as shown in Table VI.2.

Table VI.1

** Parameters of some Finite State Machines **

FSM	ni	ns	no	pti	pts	nbm
01	4	5	1	20	13	3
02	8	7	5	56	24	3
03	8	4	5	32	16	2
04	4	27	3	108	55	5
05	4	8	3	32	18	3
06	2	7	2	14	10	3
07	2	15	3	30	23	4

ni : number of input bits
ns : number of states
no : number of output bits
pti : initial cardinality of the symbolic cover
pts : minimal symbolic cover cardinality
nbm : encoding minimum length

Table VI.2

** Comparison of State Encodings between KISS and MINAS **

FSM	Encoding Length		Minimal Boolean Cover Cardinality		Gain in Silicon Area		Execution Time (seconds)	
	KISS	MINAS	KISS	MINAS	mtx	fb	KISS	MINAS
01	3	3	10	9	10%	0%	4	16
02	5	3	22	25	6%	40%	31	37
03	4	2	14	16	7%	50%	10	12
04	9	5	48	61	14%	44%	748	88
05	4	3	17	17	14%	25%	11	16
06	3	3	8	8	0%	0%	4	22
07	5	4	17	18	8%	20%	26	16

mtx : matrix = 100*(aa.KISS - aa.MINAS)/(aa.KISS)
fb : feedback = 100*(nb.KISS - nb.MINAS)/(nb.KISS)

aa : array area = (2*nb + ni + no)*pt
nb : encoding length
ni : number of input bits
no : number of output bits
pt : minimal Boolean cover cardinality

Execution time :

for MINAS: on an APOLLO computer
for KISS : on a VAX-UNIX computer

Logic Minimiser :

for MINAS: MOM
for KISS : ESPRESSO-II

VII. CONCLUSIONS

Comparing results of different programs, it is clear that the Method of Maximal Adjacencies deserves attention.

Preliminary experiments have shown that we are aware with an useful approach.

Further improvement on the program which implements our strategy can be achieved. For instance, it is possible to change the implementation of MINAS to make it run much faster.

The important point is that the algorithm is conceptually conceived to explore big finite state machines.

There are still other possibilities that emerge from the program. For instance :

< i> encoding the states with a not miminal number of bits;
< ii> exploring the trade-off between the parameters "MNSC", "MNP" and "Cost";
<iii> exploring other strategies to shape a final family of partitions.

Important aspects proposed in [1] have not been covered by this research work, particulary :

< iv> applications to unminimal machines;
< v> conditions for common terms;
< vi> dynamic ordering of adjacency conditions.

Those possibilities will be investigated and reported in a near future.

VIII. REFERENCES

[1] Jóźwiak, L.
Minimal realization of sequential machines: The method of maximal adjacencies.
Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands, 1988.
EUT Report 88-E-209

[2] De Micheli, G. and R.K. Brayton, G. Sangiovanni-Vincentelli, T. Villa
Optimal state assignment for finite state machines.
IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., Vol. CAD-4(1985), p. 269-285.

[3] Hartmanis, J. and R.E. Stearns
Algebraic structure theory of sequential machines.
Englewood Cliffs, N.J.: Prentice-Hall, 1966.
Prentice-Hall series in automatic computation

[4] Lewin, D.
Design of logic systems.
Wokingham, Berkshire, England: Van Nostrand Reinhold, 1985.

Annexe1: input and output files format definitions

Consider the following sequential machine : FSM06

(a) Next state - output table

input	present state	next state	output
10	state1	state6	00
10	state2	state5	00
10	state3	state5	00
10	state4	state6	00
10	state5	state1	10
10	state6	state1	01
10	state7	state5	00
01	state1	state4	00
01	state2	state3	00
01	state3	state7	00
01	state4	state6	10
01	state5	state2	10
01	state6	state2	01
01	state7	state6	10

(b) Input file format definition for MINAS : FSM06.DAT

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

(c) Input file format definition for MOM : FSM06.DEF

```
"State Assignment :
"State1 : 0 1 0
"State2 : 1 1 0
"State3 : 1 1 1
"State4 : 0 1 1
"State5 : 0 0 0
"State6 : 1 0 0
"State7 : 0 0 1
"
$IN I1,I2,PS2,PS1,PS0
$OUT NS2,NS1,NS0,O1,O2
"
1 0 0 1 0 | A . . . .
1 0 1 1 0 | . . . . .
1 0 1 1 1 | . . . . .
1 0 0 1 1 | A . . . .
1 0 0 0 0 | . A . A .
1 0 1 0 0 | . A . . A
1 0 0 0 1 | . . . . .
0 1 0 1 0 | . A A . .
0 1 1 1 0 | A A A . .
0 1 1 1 1 | . . A . .
0 1 0 1 1 | A . . A .
0 1 0 0 0 | A A . A .
0 1 1 0 0 | A A . . A
0 1 0 0 1 | A . . A .
0 0 0 1 0 | - - - - -
0 0 1 1 0 | - - - - -
0 0 1 1 1 | - - - - -
0 0 0 1 1 | - - - - -
0 0 0 0 0 | - - - - -
0 0 1 0 0 | - - - - -
0 0 0 0 1 | - - - - -
1 1 0 1 0 | - - - - -
1 1 1 1 0 | - - - - -
1 1 1 1 1 | - - - - -
1 1 0 1 1 | - - - - -
1 1 0 0 0 | - - - - -
1 1 1 0 0 | - - - - -
1 1 0 0 1 | - - - - -
```

(d) Output file from MOM : FSM06.MIN

```
"State Assignment :
"State1 : 0 1 0
"State2 : 1 1 0
"State3 : 1 1 1
"State4 : 0 1 1
"State5 : 0 0 0
"State6 : 1 0 0
"State7 : 0 0 1
"
$IN I1,I2,PS2,PS1,PS0
$OUT NS2,NS1,NS0,O1,O2
"

Min-cover, found 8 essential productterms
" The function contains 16 don't cares and 12 actives

"     P P P | N N N
"I I S S S  | S S S O O
"1 2 2 1 0  | 2 1 0 1 2
"___________|__________
 x 1 1 x 0  | a . . . .
 x 1 x 0 0  | a . . . .
 x 1 0 x 1  | a . . a .
 1 x 0 1 x  | a . . . .
 x x 0 0 0  | . a . a .
 0 x 1 1 x  | . . a . .
 x 1 x 1 0  | . a a . .
 x x 1 0 0  | . a . . a

"FILE: FSM06.def,
"minimized by MOM version 4.32, Multiple Output Mode
```

Annexe2 : set of industrial sequential machines

```
" FSM 01 :

   input   present  next     output
           state    state

    1000   state1   state1   1
    0100   state1   state1   1
    0010   state1   state2   1
    0001   state1   state2   1

    1000   state2   state2   1
    0100   state2   state3   1
    0010   state2   state2   1
    0001   state2   state1   1

    1000   state3   state3   1
    0100   state3   state5   1
    0010   state3   state3   1
    0001   state3   state5   1

    1000   state4   state4   1
    0100   state4   state2   1
    0010   state4   state3   1
    0001   state4   state3   1

    1000   state5   state5   1
    0100   state5   state5   1
    0010   state5   state1   1
    0001   state5   state4   1
```

"FSM 02 :

input	present state	next state	output
10000000	state1	state3	00010
10000000	state2	state1	01001
10000000	state3	state3	10010
10000000	state4	state3	00010
10000000	state5	state1	01001
10000000	state6	state1	01001
10000000	state7	state3	10010
01000000	state2	state2	01001
01000000	state5	state2	01001
01000000	state6	state2	01001
01000000	state1	state4	00010
01000000	state3	state4	10010
01000000	state4	state4	00010
01000000	state7	state4	10010
00100000	state5	state1	10001
00100000	state6	state1	10001
00100000	state7	state1	10001
00100000	state1	state3	01010
00100000	state2	state3	00100
00100000	state3	state3	01010
00100000	state4	state3	00100
00010000	state5	state1	10101
00010000	state6	state1	10101
00010000	state7	state1	10101
00010000	state1	state4	01010
00010000	state3	state4	01010
00010000	state2	state5	00100
00010000	state4	state5	00100
00001000	state2	state2	00101
00001000	state5	state2	00101
00001000	state1	state3	01000
00001000	state3	state3	01000
00001000	state4	state3	10100
00001000	state6	state3	10100
00001000	state7	state3	10100
00000100	state2	state1	00101
00000100	state5	state1	00101
00000100	state1	state5	00010
00000100	state3	state5	10010
00000100	state4	state5	00010
00000100	state6	state5	10100
00000100	state7	state5	10010

00000010	state2	state1	00001
00000010	state5	state2	10001
00000010	state6	state2	10001
00000010	state7	state2	10001
00000010	state1	state5	01010
00000010	state3	state5	01010
00000010	state4	state5	10100
00000001	state2	state2	00001
00000001	state5	state2	10101
00000001	state6	state2	10101
00000001	state7	state2	10101
00000001	state1	state6	01000
00000001	state3	state6	01000
00000001	state4	state7	10000

"FSM 03 :

input	present state	next state	output
10000000	state1	state1	00101
10000000	state2	state2	10010
10000000	state3	state1	00101
10000000	state4	state2	10010
01000000	state1	state2	00010
01000000	state2	state2	10100
01000000	state3	state2	00010
01000000	state4	state2	10100
00100000	state1	state3	00010
00100000	state2	state3	10010
00100000	state3	state3	00010
00100000	state4	state3	10010
00010000	state3	state1	00100
00010000	state4	state1	00100
00010000	state1	state2	10001
00010000	state2	state2	10001
00001000	state3	state1	00100
00001000	state4	state1	00100
00001000	state1	state3	10101
00001000	state2	state3	10101
00000100	state1	state1	01001
00000100	state3	state1	10100
00000100	state4	state1	01001
00000100	state2	state3	01001
00000010	state1	state2	01010
00000010	state2	state2	01010
00000010	state3	state2	01000
00000010	state4	state2	01010
00000001	state1	state3	01010
00000001	state2	state3	01010
00000001	state4	state3	10000
00000001	state3	state4	01010

"FSM04 :

input	present state	next state	output
1000	state1	state3	001
1000	state2	state1	001
1000	state3	state4	001
1000	state4	state4	010
1000	state5	state1	010
1000	state6	state3	010
1000	state7	state9	010
1000	state8	state15	010
1000	state9	state1	000
1000	state10	state14	000
1000	state11	state3	000
1000	state12	state20	000
1000	state13	state3	101
1000	state14	state1	101
1000	state15	state4	101
1000	state16	state20	000
1000	state17	state15	010
1000	state18	state4	100
1000	state19	state18	100
1000	state20	state19	100
1000	state21	state2	100
1000	state22	state3	000
1000	state23	state2	100
1000	state24	state14	000
1000	state25	state15	010
1000	state26	state20	000
1000	state27	state15	010
0100	state1	state10	001
0100	state2	state2	001
0100	state3	state5	001
0100	state4	state5	010
0100	state5	state2	010
0100	state6	state21	010
0100	state7	state18	010
0100	state8	state26	000
0100	state9	state5	000
0100	state10	state13	000
0100	state11	state23	000
0100	state12	state19	000
0100	state13	state10	101
0100	state14	state2	101
0100	state15	state5	101
0100	state16	state19	000
0100	state17	state23	000
0100	state18	state5	010
0100	state19	state23	010
0100	state20	state20	010
0100	state21	state1	010
0100	state22	state3	010
0100	state23	state1	010
0100	state24	state13	000
0100	state25	state3	010
0100	state26	state19	000
0100	state27	state3	010

0010	state1	state11	001
0010	state2	state8	001
0010	state3	state6	001
0010	state4	state6	010
0010	state5	state16	010
0010	state6	state10	010
0010	state7	state19	010
0010	state8	state13	010
0010	state9	state6	000
0010	state10	state1	000
0010	state11	state24	000
0010	state12	state18	000
0010	state13	state11	101
0010	state14	state8	101
0010	state15	state6	101
0010	state16	state13	010
0010	state17	state18	000
0010	state18	state6	100
0010	state19	state24	100
0010	state20	state9	100
0010	state21	state13	100
0010	state22	state15	100
0010	state23	state13	010
0010	state24	state13	100
0010	state25	state15	000
0010	state26	state18	000
0010	state27	state13	100

0001	state1	state12	001
0001	state2	state9	001
0001	state3	state7	001
0001	state4	state7	010
0001	state5	state17	010
0001	state6	state22	010
0001	state7	state20	010
0001	state8	state14	010
0001	state9	state7	000
0001	state10	state2	000
0001	state11	state25	000
0001	state12	state15	000
0001	state13	state12	101
0001	state14	state9	101
0001	state15	state7	101
0001	state16	state14	010
0001	state17	state27	000
0001	state18	state7	100
0001	state19	state25	100
0001	state20	state26	100
0001	state21	state14	100
0001	state22	state15	000
0001	state23	state14	010
0001	state24	state14	100
0001	state25	state15	000
0001	state26	state21	000
0001	state27	state14	100

" FSM 05 :

input	present state	next state	output
1000	state1	state1	001
1000	state2	state3	000
1000	state3	state1	001
1000	state4	state4	100
1000	state5	state3	000
1000	state6	state7	000
1000	state7	state4	010
1000	state8	state4	100
0100	state3	state1	101
0100	state7	state1	101
0100	state1	state4	010
0100	state2	state4	000
0100	state5	state4	100
0100	state4	state5	101
0100	state8	state5	100
0100	state6	state8	000
0010	state1	state2	001
0010	state3	state2	001
0010	state2	state3	010
0010	state5	state3	010
0010	state6	state3	010
0010	state8	state3	010
0010	state4	state4	010
0010	state7	state5	010
0001	state3	state2	101
0001	state7	state2	101
0001	state5	state3	100
0001	state6	state3	100
0001	state8	state3	100
0001	state1	state5	010
0001	state4	state5	101
0001	state2	state6	000

" FSM06 :

input	present state	next state	output
10	state1	state6	00
01	state1	state4	00
10	state2	state5	00
01	state2	state3	00
10	state3	state5	00
01	state3	state7	00
10	state4	state6	00
01	state4	state6	10
10	state5	state1	10
01	state5	state2	10
10	state6	state1	01
01	state6	state2	01
10	state7	state5	00
01	state7	state6	10

" FSM 07 :

input	present state	next state	output
10	state1	state8	000
10	state2	state4	000
10	state3	state5	000
10	state4	state8	000
10	state5	state8	000
10	state6	state13	000
10	state7	state4	000
10	state8	state1	001
10	state9	state4	000
10	state10	state1	010
10	state11	state3	010
10	state12	state4	100
10	state13	state5	100
10	state14	state3	100
10	state15	state4	000
01	state1	state9	000
01	state2	state3	000
01	state3	state6	000
01	state4	state11	000
01	state5	state12	000
01	state6	state14	000
01	state7	state15	000
01	state8	state2	001
01	state9	state3	001
01	state10	state2	010
01	state11	state4	010
01	state12	state3	001
01	state13	state6	100
01	state14	state7	100
01	state15	state6	000

ISSN 0167-9708
Coden: TEUEDE

(188) Jóźwiak, J.
THE FULL DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE STATE AND OUTPUT BEHAVIOUR REALIZATION.
EUT Report 88-E-188. 1988. ISBN 90-6144-188-9

(189) Pineda de Gyvez, J.
ALWAYS: A system for wafer yield analysis.
EUT Report 88-E-189. 1988. ISBN 90-6144-189-7

(190) Siuzdak, J.
OPTICAL COUPLERS FOR COHERENT OPTICAL PHASE DIVERSITY SYSTEMS.
EUT Report 88-E-190. 1988. ISBN 90-6144-190-0

(191) Bastiaans, M.J.
LOCAL-FREQUENCY DESCRIPTION OF OPTICAL SIGNALS AND SYSTEMS.
EUT Report 88-E-191. 1988. ISBN 90-6144-191-9

(192) Worm, S.C.J.
A MULTI-FREQUENCY ANTENNA SYSTEM FOR PROPAGATION EXPERIMENTS WITH THE OLYMPUS SATELLITE.
EUT Report 88-E-192. 1988. ISBN 90-6144-192-7

(193) 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.
EUT Report 88-E-193. 1988. ISBN 90-6144-193-5

(194) Hosselet, L.M.L.F.
MARTINUS VAN MARUM: A Dutch scientist in a revolutionary time.
EUT Report 88-E-194. 1988. ISBN 90-6144-194-3

(195) Bondarev, V.N.
ON SYSTEM IDENTIFICATION USING PULSE-FREQUENCY MODULATED SIGNALS.
EUT Report 88-E-195. 1988. ISBN 90-6144-195-1

(196) Liu Wen-Jiang, Zhu Yu-Cai and Cai Da-Wei
MODEL BUILDING FOR AN INGOT HEATING PROCESS: Physical modelling approach and identification approach.
EUT Report 88-E-196. 1988. ISBN 90-6144-196-X

(197) Liu Wen-Jiang and Ye Dau-Hua
A NEW METHOD FOR DYNAMIC HUNTING EXTREMUM CONTROL, BASED ON COMPARISON OF MEASURED AND ESTIMATED VALUE.
EUT Report 88-E-197. 1988. ISBN 90-6144-197-8

(198) Liu Wen-Jiang
AN EXTREMUM HUNTING METHOD USING PSEUDO RANDOM BINARY SIGNAL.
EUT Report 88-E-198. 1988. ISBN 90-6144-198-6

(199) Jóźwiak, L.
THE FULL DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE OUTPUT BEHAVIOUR REALIZATION.
EUT Report 88-E-199. 1988. ISBN 90-6144-199-4

(200) 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.
EUT Report 88-E-200. 1988. ISBN 90-6144-200-1

(201) Woudenberg, H. van and R. van den Born
HARDWARE SYNTHESIS WITH THE AID OF DYNAMIC PROGRAMMING.
EUT Report 88-E-201. 1988. ISBN 90-6144-201-X

(202) Engelshoven, R.J. van and R. van den Born
COST CALCULATION FOR INCREMENTAL HARDWARE SYNTHESIS.
EUT Report 88-E-202. 1988. ISBN 90-6144-202-8

(203) Delissen, J.G.M.
THE LINEAR REGRESSION MODEL: Model structure selection and biased estimators.
EUT Report 88-E-203. 1988. ISBN 90-6144-203-6

(204) Kalasek, V.K.I.
COMPARISON OF AN ANALYTICAL STUDY AND EMTP IMPLEMENTATION OF COMPLICATED THREE-PHASE SCHEMES FOR REACTOR INTERRUPTION.
EUT Report 88-E-204. 1988. ISBN 90-6144-204-4

ISSN 0167-9708
Coden: TEUEDE

(205) Butterweck, H.J. and J.H.F. Ritzerfeld, M.J. Werter
FINITE WORDLENGTH EFFECTS IN DIGITAL FILTERS: A review.
EUT Report 88-E-205. 1988. ISBN 90-6144-205-2

(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 TWONFIL AND EMTP.
EUT Report 88-E-206. 1988. ISBN 90-6144-206-0

(207) Schuurman, W. and M.P.H. Weenink
STABILITY OF A TAYLOR-RELAXED CYLINDRICAL PLASMA SEPARATED FROM THE WALL BY A VACUUM LAYER.
EUT Report 88-E-207. 1988. ISBN 90-6144-207-9

(208) Lucassen, F.H.R. and H.H. van de Ven
A NOTATION CONVENTION IN RIGID ROBOT MODELLING.
EUT Report 88-E-208. 1988. ISBN 90-6144-208-7

(209) Jóźwiak, L.
MINIMAL REALIZATION OF SEQUENTIAL MACHINES: The method of maximal adjacencies.
EUT Report 88-E-209. 1988. ISBN 90-6144-209-5

(210) Lucassen, F.H.R. and H.H. van de Ven
OPTIMAL BODY FIXED COORDINATE SYSTEMS IN NEWTON/EULER MODELLING.
EUT Report 88-E-210. 1988. ISBN 90-6144-210-9

(211) Boom, A.J.J. van den
H_∞-CONTROL: An exploratory study.
EUT Report 88-E-211. 1988. ISBN 90-6144-211-7

(212) Zhu Yu-Cai
ON THE ROBUST STABILITY OF MIMO LINEAR FEEDBACK SYSTEMS.
EUT Report 88-E-212. 1988. ISBN 90-6144-212-5

(213) Zhu Yu-Cai, M.H. Driessen, A.A.H. Damen and P. Eykhoff
A NEW SCHEME FOR IDENTIFICATION AND CONTROL.
EUT Report 88-E-213. 1988. ISBN 90-6144-213-3

(214) Bollen, M.H.J. and G.A.P. Jacobs
IMPLEMENTATION OF AN ALGORITHM FOR TRAVELLING-WAVE-BASED DIRECTIONAL DETECTION.
EUT Report 89-E-214. 1989. ISBN 90-6144-214-1

(215) Hoeijmakers, M.J. en J.M. Vleeshouwers
EEN MODEL VAN DE SYNCHRONE MACHINE MET GELIJKRICHTER, GESCHIKT VOOR REGELDOELEINDEN.
EUT Report 89-E-215. 1989. ISBN 90-6144-215-X

(216) Pineda de Gyvez, J.
LASER: A LAyout Sensitivity ExploreR. Report and user's manual.
EUT Report 89-E-216. 1989. ISBN 90-6144-216-8

(217) Duarte, J.L.
MINAS: An algorithm for systematic state assignment of sequential machines - computational aspects and results.
EUT Report 89-E-217. 1989. ISBN 90-6144-217-6