ESKISS : a program for optimal state assignment

Research Report

Eindhoven
University of Technology
Netherlands

Department of Electrical Engineering

ESKISS: A Program for Optimal State Assignment

by
C.A.M. Oerlemans
and
J.F.M. Theeuwen

EUT Report 87-E-167
ISBN 90-6144-167-6
ISSN 0167-9708

January 1987

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Eindhoven The Netherlands

ESKISS:
A program for optimal state assignment

by

C.A.M. Oerlemans

and

J.F.M. Theeuwen

EUT Report 87-E-167

ISBN 90-6144-167-6

ISSN 0167-9708

Coden: TEUEDE

Eindhoven

January 1987

COOPERATIVE DEVELOPMENT OF AN INTEGRATED, HIERARCHICAL AND MULTIVIEW VLSI-DESIGN SYSTEM WITH DISTRIBUTED MANAGEMENT ON WORKSTATIONS.

(Multiview VLSI-design System ICD)

code: 991

DELIVERABLE

Report on activity 5.2.A: Review of the state encoding problem.
Study of relations between state encoding and logic minimisation.

Abstract:
Solving the problem of optimal state assignment solves a variety of problems involving *controllers, as they can effectively be realized using a* Finite State Machine (FSM). Assigning the right codes to the right states of the FSM is very important, as it strongly affects the size of the implementation of the FSM.

In this report an algorithm is presented, *eskiss*, that searches for an optimal encoding of the states with regard to the needed size of a PLA to implement the FSM. The description of the FSM is given by means of a *symbolic cover* (state transition table) and is firstly minimized by the symbolic minimizer ESPRESSO.

Eskiss generates *state groups*, containing states that show identical transitions in the transition table. Based on these state groups a relation can be stated, such that if the encodings of the states satisfy the relation, the assignment has been optimal with regard to the size of the PLA. This relation is called the *constraint relation*. Eskiss generates an encoding that satisfies this constraint relation.

deliverable code: WP 5, *task*: 5.2, *activity*: 5.2.A.

date:	01 - 01 - 1987
partner:	Eindhoven University of Technology
author:	C.A.M. Oerlemans, J.F.M. Theeuwen.

This report was accepted as a M.Sc. Thesis of C.A.M. Oerlemans by Prof. Dr.-Ing. J.A.G. Jess, Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology. The work was performed in the period from 5 May 1986 to 18 December 1986 and was supervised by Dr.ir. J.F.M. Theeuwen.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Oerlemans, C.A.M.

ESKISS: a program for optimal state assignment / by C.A.M. Oerlemans and J.F.M. Theeuwen. - Eindhoven: University of Technology. - Fig., tab. - (Eindhoven University of Technology research reports / Department of Electrical Engineering, ISSN 0167-9708; 87-E-167)
Met lit. opg., reg.
ISBN 90-6144-167-6
SISO 664.3 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.

CONTENTS

1. INTRODUCTION

Recent developments in integrated circuit technology enable designers to invest all their creative ambition in designing systems that may well be called "gigantic artefacts". With the current technology of Very Large Scale Integration (VLSI), the element that constrains the extent of a chip design is the *simple* mind of the designer, rather than the *ingenious* equipment of the manufacturer.

With a complexity of hundreds of thousands of components on a chip, it is impossible for man to design such a circuit by hand or by means of a layout editor. Firstly, because it would be too time-consuming to design all of the components by hand and secondly, because that amount of components will increase the probability of a design error to almost 100%.

It would therefore be useful to have a system that accepts as input a *functional description* of the circuit to be designed, making it possible to describe, without errors, exactly what the designers wishes are, without bothering about implementation details. The output of such a system should be a *detailed layout* of the chip, optimized as far as possible and ready to be physically implemented.

A system (or program) that satisfies these conditions is called a *silicon compiler* and is a subject of many recent research projects. One such project is the development of a system for *high level design*, a project of the department of electrical engineering, laboratory automatic system design at the Eindhoven University of Technology. The functional description of the circuit is given by means of a high level language like Pascal or LISP and is translated (*compiled* !) into a file describing the physical layout of the circuit.

The concept of the generated layout is the following: An *N-stage Finite State Machine* (FSM) controls a number of logic and arithmetic modules like AND/OR gates, adders, multipliers, multiplexers, I/O registers, etc. These modules are standard designs and are collected into a standard library. The FSM is also a standard design, except for the implementation of the combinational component, which depends upon the function of the FSM and upon the *state assignment* that was chosen. (The interested reader is referred to [7] for further details about this project.)

As indicated in the above paragraph, the chosen *state assignment* affects the form of the combinational component of the FSM. Further investigation of some examples shows, that the form of the combinational component can vary very much when different assignments are tried. It is therefore very useful to search for an assignment that can be regarded as being at least suboptimal with respect to the form of the combinational component.

A simple calculation shows, that the number of possible assignments grows with the number of stages (states of the FSM) to astronomic figures and searching for the best assignment cannot be done by simply trying various possible solutions. The algorithm that has to be found, must search for a (sub-)optimal solution in a heuristic way, in order to keep the processing time within reasonable boundaries.

This report gives the results of a graduation traineeship in which an algorithm for finding a good assignment is discussed and implemented. The system on which the implementation runs is a HP9000 system with UNIX operating system. The algorithm has been implemented in C. The program uses an existing *logic minimizer* called *ESPRESSO*; documentation about *ESPRESSO* can be found in one of the appendices.

A theoretical approach to the problem is the goal of chapter 2, where formal definitions and some theorems are stated, while the next chapter discusses the concept and the details of the encoding algorithm. In chapter 4 the implementation in C is given and chapters 5 and 6 give some tests and conclusions of the implemented algorithm.

In one of the appendices a marginal aspect of the FSM is discussed. Apart from the silicon compiler, it would be useful to be able to enter the description of a FSM by hand and let the system search for an optimal assignment for that FSM. It is therefore needed to define an input language to describe a FSM in a way that suits the way of thinking of the designer. When such a language exists, a compiler for that language can be made with little effort. This sub-project is still under investigation.

2. THEORETICAL APPROACH

This chapter tries to give the reader some definitions in order to understand the remainder of the report. Firstly, the concept of the hardware implementation of a finite state machine is discussed and secondly, the principles of symbolic covers and minimization are stated in a way that will be used throughout the report. The final topic of this chapter explains the method used to find an optimal state assignment: *constrained state encoding*. Note that this is not an algorithm, but merely a concept that forms the basis for it. The algorithm itself is discussed in the next chapter.

This chapter is fully derived from [1] and more information about the topics of this chapter can be found there. Not to bore the reader with details about exact proves of certain theorems, all these cases are omitted and again, the interested reader is referred to [1].

2.1 Hardware implementation

Before something senseful can be said about the criteria that form the basis for the algorithm to assign the optimal codes to the states of the finite state machine, the hardware implementation needs discussion. The concept for the hardware of the FSM is shown in figure 2.1.

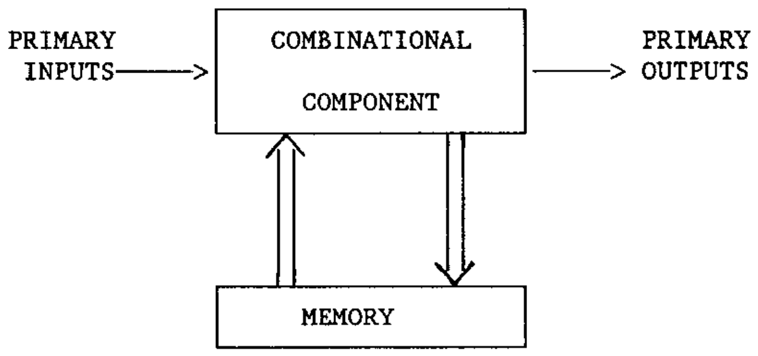

Figure 2.1. Hardware concept for FSM.

In chapter 1 the term *combinational component* was already used. Here the term is given more attention, as it is an important part of the FSM. It is the combinational component that will take most of the space on the chip, especially when large FSM's are regarded. This part of the design should therefore get good attention in the optimization phase and a well suited form of implementation must be found for it.

To the designer the choice of implementing the combinational component by means of several different technologies lies open and he should remember, that large designs cannot be made by hand, but need a computer. Therefore, the choice that is made in this report, based on the assumption that a structured design method needs a structured implementation technology, is the implementation of the combinational component by means of a PLA. Other implementations, such as separate AND-, OR-, INVERT-gates, are possible, but do not have the same regular structure as the PLA has. Usually, if the states are encoded such that the PLA implementation is optimal, the other implementations also show relatively good results, so the PLA is a good estimation for the other implementations.

The other part of the FSM is the *memory component*. This part may be implemented by several forms of memory elements: *Delay-latches*, *toggle-flipflops*, *JK-flipflops*, etc. The choice of one of these elements is mostly arbitrary, here the *delay-latch* (*D-latch*) will be used.

2.2 Symbolic cover and symbolic minimization

The terms *symbolic cover* and *symbolic minimization* apply to the way the problem of state encoding will be tackled. The description of the FSM must be symbolic, i.e. the states, the values of the inputs and the values of the outputs must be names instead of (binary, hexadecimal) values as they will appear in the hardware configuration. Then, before assigning codes to the states, symbolic minimization is applied to the symbolic definition of the FSM. A symbolic minimizer is a logic minimizer that accepts names instead of values for the values of the variables. The symbolic minimizer used is called *ESPRESSO* and will be addressed by that name in the remainder of this report.

The symbolic definition of the FSM is a symbolic cover as it *covers* the definition of the FSM in a *symbolic* representation. The symbolic cover is formally stated by the 5-tuple $(X, Y, Z, \delta, \lambda)$ where:

$X = \{x_1, x_2, \dots, x_{\lvert X \rvert}\}$	*set of primary input symbols,*
$Y = \{y_1, y_2, \dots, y_{\lvert Y \rvert}\}$	*set of internal states,*
$Z = \{z_1, z_2, \dots, z_{\lvert Z \rvert}\}$	*set of primary output symbols,*
$\delta: X \times Y \rightarrow Y$	*next state function,*
$\lambda: X \times Y \rightarrow Z \; (\lambda: Y \rightarrow Z)$	*output function for a Mealy (Moore) machine [6].*

The symbolic cover is usually given in the form of a state transition table. Then, the functions δ and λ are specified by giving exactly for each possible input symbol and each possible old state, the new state (δ) and the output symbol (λ). The following example (from [1]) should give a clearer insight.

Each row of the state transition table is called an implicant. For instance, the implicant *zero zero START state6* shows, that with input "*zero*", state "*START*" is transformed into "*state6*", asserting "*zero*" as output. All other implicants have a similar meaning and thus, the symbolic description of the FSM is completely covered by this table.

in	out	old	new
zero	zero	START	state6
zero	zero	state2	state5
zero	zero	state3	state5
zero	zero	state4	state6
zero	two	state5	START
zero	one	state6	START
zero	zero	state7	state5
one	zero	START	state4
one	zero	state2	state3
one	zero	state3	state7
one	two	state4	state6
one	two	state5	state2
one	one	state6	state2
one	two	state7	state6

Now, this symbolic cover can be minimized by the symbolic minimizer ESPRESSO. Note, that according to appendix 1, the input to ESPRESSO does not agree with the form of the table above. For the sake of simplicity, the transition table will be as is shown above and when used as input to ESPRESSO it will first be formated into the right form, according to appendix 1.

ESPRESSO will give the following output:

```
label START state6 state2 state5 state3 state4 state7

                 10 0101000 1000000000
                 01 0101000 0010000000
                 01 0010000 0000100100
                 01 1000000 0000010100
                 01 0000100 0000001100
                 01 0000011 0100000010
                 10 1000010 0100000100
                 11 0100000 0000000001
                 10 0010101 0001000100
                 11 0001000 0000000010
```

To understand this result, the term "one-hot-encoding" must be explained. One-hot-encoding will encode a certain variable using a row of bits. This kind of encoding needs as many bits as there are symbols to be encoded and each bit corresponds to one of these symbols. Thus, a row of bits indicates which symbols are present in that implicant: 1 indicates corresponding symbol is present, 0 indicates not present. If a row of bits contains zero entries only, no symbolic value of the corresponding variable is indicated, meaning that nothing can be said about the actual value. Non-zero entries only indicates the don't care value. Note, that the label header indicates which symbolic statename the respective bits correspond to.

In the above example, the third implicant *01 0010000 0000100100* indicates, that with input *"01"* (=*one*) state *"0010000"* (=*state2*) is transformed into *"0000100"* (=*state3*), asserting *"100"* (=*zero*) as output. This is the same implicant as the ninth in the transition table. Likewise, the sixth implicant indicates, that with input *one* the states *state4* and *state7* are transformed into *state6*, asserting *two* as output. The corresponding implicants in the transition table are the eleventh and the fourteenth. Thus, the symbolic cover is reduced to ten implicants, instead of fourteen in the transition table.

Now, before discussing the constrained state encoding problem, two more definitions are given. From the minimized symbolic cover state subsets can be composed, having more than one element and corresponding to the column with old states from one of the implicants. Such subset is called a *state group*. Given a state assignment and a state group, the corresponding *group face* (or simply *face*) is the minimal dimension subspace containing the encodings of the states assigned to that group [1]. The constrained state encoding problem is now defined as follows:

> *Given a set of state groups, find an encoding such that each group face does not intersect the code assigned to any state not contained in the corresponding group.*

In [1] theorems are formulated, indicating that a solution to the constrained state encoding problem is a suboptimal solution to the state encoding problem using a PLA. This constrained state encoding problem is used to construct an algorithm to solve the state encoding problem. The next section will show a way to determine whether a given state encoding is a solution or not and the next chapter will use the results of the next section to construct the algorithm.

2.3 Constrained state encoding

This section discusses a relation, that the state encodings have to satisfy, in order to be a solution to the constrained state encoding problem. This relation is called the *constraint relation* and is derived from the set of state groups, defined in the previous section.

The discussion of the constraint relation needs a few more definitions to state the relation in a mathematical manner. All the definitions are equal to those given in [1], but all theorems and proves are omitted.

Firstly, an extension to the type Boolean is needed. *Pseudo-Boolean* entries are elements from {0, 1, *, ϕ}, where * denotes the don't care value (either 1 or 0) and ϕ the empty value (neither 1 nor 0). Pseudo-Booleans can be subject to two different operations, *conjunction* ($\wedge$) and *disjunction* ($\vee$), defined as follows:

conjunction

$\wedge$	0	1	*	ϕ
0	0	ϕ	0	ϕ
1	ϕ	1	1	ϕ
*	0	1	*	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ

disjunction

$\vee$	0	1	*	ϕ
0	0	*	*	0
1	*	1	*	1
*	*	*	*	*
ϕ	0	1	*	ϕ

The state groups of the minimized symbolic cover will be grouped together to form the *constraint matrix A*:

$$A = \begin{bmatrix} a_{1\cdot} \\ a_{2\cdot} \\ \cdots \\ a_{n_l\cdot} \end{bmatrix} = [a_{\cdot 1} \mid a_{\cdot 2} \mid \cdots \mid a_{\cdot n_s}]$$

where n_l is the number of state groups and n_s the number of states. State j belongs to group i if $a_{ij} = 1$.

A row of the constraint matrix is said to be a *meet* if it represents the bit-wise conjunction of two or more state groups. A row of the constraint matrix is said to be a *prime* if it is not a meet.

The *state code matrix* S is a matrix whose rows are state encodings:

$$S = \begin{bmatrix} s_{1\cdot} \\ s_{2\cdot} \\ \cdots \\ s_{n_s\cdot} \end{bmatrix}$$

The constrained state encoding problem is to determine the state code matrix S, given a constraint matrix A.

Another operator needed to explain the constraint relation is the *selection* of b according to a, with $a \in \{0, 1\}$ and $b \in \{0, 1, *, \phi\}$:

$$a \cdot b = \begin{cases} b & \text{if } a = 1 \\ \phi & \text{if } a = 0 \end{cases}$$

Selection can be extended to two dimensional arrays and is similar to matrix multiplication, with "plus" substituted by "disjunction" and "times" by "selection".

The *face matrix* is the matrix whose rows are the group faces. It can be obtained by performing the selection of S according to the constraint matrix A:

$$F = A \cdot S$$

Let $\bar{A}$ be the matrix A with all entries complemented. Then

$$\bar{F}^i = \bar{a}_{\cdot i} \cdot s_{i\cdot}$$

is a matrix whose rows are the encoding of state i if state i does not belong to group j, else are empty values. A state encoding matrix S is a solution of the constrained state encoding problem if it satisfies the *constraint relation* for a given constraint matrix A:

$$\bar{F}^i \wedge F = \begin{bmatrix} \bar{f}^i_{1\cdot} & \wedge & f_{1\cdot} \\ \bar{f}^i_{2\cdot} & \wedge & f_{2\cdot} \\ & \cdots & \\ \bar{f}^i_{n_l\cdot} & \wedge & f_{n_l\cdot} \end{bmatrix} = \Phi, \qquad i = 1, 2, \dots, n_s$$

where Φ is the empty matrix, i.e., a matrix whose rows have at least one ϕ entry.

Now the *optimal constrained state encoding problem* can be formally stated as follows:

> *Find a state code matrix S with minimal number of columns that satisfies the constraint relation for the constraint matrix A, containing the state groups.*

Based on this statement of the problem, an algorithm for state encoding will be discussed in the next chapter. The algorithm will be explained and its consistency with the constrained state encoding problem will be shown, without using complex theorems.

3. STATE ENCODING ALGORITHM

State encoding is a complex problem that can only be (partly) solved by means of a heuristic strategy. An algorithm using this strategy will determine a suboptimal state encoding matrix for a given finite state machine.

Here, an algorithm using a heuristic strategy called *constrained state encoding* is presented, based on the theory from the previous chapter. The algorithm and its theoretical basis were firstly presented in [1]. A state encoding matrix is said to be a solution of the constrained state encoding problem, if it satisfies a certain relation, called the *constraint relation*.

3.1 The structure of the algorithm

Suppose the FSM has n_s states and ESPRESSO has generated a constraint matrix A, containing only prime rows (see chapter 2). Then, state encoding will be performed by considering one state or state subset at a time. An optimal encoding for this state(set) is sought for and if found, added to the state encoding matrix. If an encoding cannot be found, the dimension of the state code subspace is increased, by adding one column to the state encoding matrix. Then again, an optimal encoding is sought for.

The process of seeking for an optimal encoding is based on the assumption, that an optimal state encoding will satisfy the constraint relation. Each candidate from the state code subspace of current dimension n_b is tested against this relation and from those that satisfy it, the most suitable one is selected according to a heuristic strategy, explained in section 3.2. Then the next state(set) is selected and encoded, until all states have been considered.

The structure of the algorithm is the following:

1: *Select an uncoded state (or state subset)*;
2: *Determine the candidate encodings for that state(s), satisfying the constraint relation*;
3: *If no candidate exists, increase the state code dimension by one and go to 2*;
4: *Select from the candidates of encodings the most suitable one and add it to the state encoding matrix*;
5: *If all states have been encoded, stop. Else go to 1*;

Only one theorem from [1] will be used to discuss the concept of the algorithm, leaving the proof to the enthusiastic reader.

Let S satisfy the constraint relation for a given A. Then there exists a matrix

$$S' = \begin{bmatrix} S \mid R \mid T \\ \sigma \end{bmatrix} \qquad R \in \{0,\ 1\}^{n_s \times n_p} \qquad T \in \{0\}^{n_s \times n}$$

that satisfies the constraint relation for

$$A' = \begin{bmatrix} A \mid \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \end{bmatrix}$$

where n_p is the number of rows of A' with non-zero entries in one of the columns $\alpha_1, \alpha_2, \cdots, \alpha_n$. In [1] the theorem is proved for R is A^{1T}, where A^1 is the matrix of rows of A having a non-zero entry in one of the columns $\alpha_1, \alpha_2, \cdots, \alpha_n$.

In simple words the theorem says, that an encoding for the next stateset with corresponding columns $\alpha_1, \alpha_2, \cdots, \alpha_n$, can always be found within a finite number of iterations. Adding as many columns with zero entries only as there are states in the stateset and adding the columns of R one at a time, will always reshape the state encoding matrix in a way that will allow σ to be a candidate for the state(s).

3.2 The algorithm in detail

Now, the algorithm is presented in more detail, described in a meta-language looking like C, the programming language used to implement the algorithm in the next chapter. The set Σ is the next stateset to be encoded, Γ is the set of already encoded states. The input to the algorithm is the constraint matrix A, while each iteration uses a matrix A', containing (in the right order) the columns of A, corresponding to the states in Γ. S is the state encoding matrix (seen as a set of state encodings) and σ is the selected candidate(set).

```
$S = \emptyset$;
$\Gamma = \emptyset$;
$A$ = compress($A$);
do {
        $\Sigma$ = state-select;
        $A' = \left[ A' \mid a_{\cdot\Sigma} \right]$;
        $\Gamma = \Gamma \cup \Sigma$;
        $C = \emptyset$;
        while ($C = \emptyset$) {
            $C$ = candidates($S$, $A'$);
            if ($C = \emptyset$) adjoin($S$);
            else        $\sigma$ = code-select($C$);
        }
        $S = \begin{bmatrix} S \\ \sigma \end{bmatrix}$;
} while ($\Gamma$ is a proper subset of the stateset);
```

procedure compress(A) returns the prime rows of A. Procedure state-select sorts the states according to a heuristic strategy and returns the current state (or state subset) Σ to be encoded. More about the state-select strategy can be found below. The constraint matrix A' represents a permutation of the columns of A corresponding to the encoded and selected states in the given order.

Procedure **candidates**(S, A') returns the set of encodings that can be assigned to Σ, of the same length as those represented by S. In particular: $C = \{c$ such that $[{S \atop c}]$ satisfies the constraint relation for $A'\}$. Note that C may be empty.

The **code-select** routine returns an element of C according to a heuristic criterion. Let $u(\sigma)$ be the minimum number of codes covered by one face from the face matrix $A \cdot S$, i.e.

$$u(\sigma) = \min_{\textit{all faces}} (2^{\text{number of } *}) \qquad \text{with } * \text{ is the don't care value and a face is a row of the face matrix}$$

Then $u(\sigma)/2^{n_b}$ represents the "utilization" of the Boolean space of current size n_b. The higher the utilization of the Boolean space, the higher the probability is, that C will be empty at the next iteration of the algorithm and that n_b has to be increased. Since encodings are selected so that the final code length is as short as possible, σ is chosen as: $\sigma = \arg\min u(\sigma)$.

Procedure **adjoin**(S) is invoked when the candidateset is empty and the code space dimension has to be increased. Let $T = \{0\}^{n_s \times 1}$, i.e. a column of n_s zeros. Note that T is not a column of zeros with length equal to the number of already encoded states. Therefore, if by coincidence a column of zeros would appear in the state encoding matrix, this column is *not* equal to T ! Let S be the subset of the columns of S different from T and let t be the number of columns of S equal to T. Let $R = A^{1T}$, where A^1 is the subset of prime rows of A having a nonzero entry in one of the columns $a_{\cdot\Sigma}$. This matrix R already appeared in the theorem of section 3.1. Then:

```
    adjoin($S$)
    if ($t < |\Sigma|$) return([$S|T$]);
    else {
        $R'$ = set of columns of $R$ not already adjoined to $S$;
        $r$ = column of $R'$ with minimal 1-count;
        return([$S|r$]);
    };
```

It will be apparent, that for each Σ there exists a typical R and R', different from the R and R' of the other Σ's. At each invocation of **adjoin**, for the same Σ, the size of the matrix R' may decrease, while t may increase or decrease, depending on the current value of t and $|\Sigma|$.

It should be clear now, what the meaning of the theorem of section 3.1 is: After a finite number of iterations all the columns of R and as many columns of zeros as the cardinality of the selected stateset will be added to the state encoding matrix. Then, the candidateset C will not be empty. Most of the time C will contain elements *before* all the above indicated columns are added. This and the structure of the algorithm make it apparent, that the algorithm will always terminate in a finite number of iterations and will construct a state encoding matrix satisfying the constraint relation.

As mentioned above, **state-select** returns the next state(set) to be encoded. Which state(set) will be returned is determined by a heuristic strategy, that will be discussed below.

First another definition:

> A *dominating set* is a maximal cardinality set of states, such that no state *dominates* any other state in the set. State p *dominates* state q if $a_{ip} \geq a_{iq}$ for $i = 1, 2, \cdots, n_l$, where n_l is the number of state groups. Note that a dominating set is not necessarily unique.

The procedure **state-select** will now select the next state(set) according to the following strategy: At the first invocation a dominating set will be returned. Since in general a dominating set contains but a fraction of the states in the state set, encoding a dominating set can be done within reasonable time. At the second and further invocations one state is returned, being the state with the highest column 1-count in the constraint matrix *A*. According to [1], this strategy assures, that a state encoding matrix satisfying the constraint relation will be found, with $n_b \le n_s$, where n_b is the number of bits per code, i.e. the code length.

As an example the constraint matrix from section 2.2 is used as input to the algorithm. compress returns the matrix

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

where the columns of *A* correspond to the states *START*, *state6* *state2*, *state5*, *state3*, *state4*, *state7* respectively.

A dominating set is {*state7*, *START*, *state5*} and the first time candidates is called, *C* is empty, as $n_b = 0$. After two times adjoining, $n_b = 2$ and code-select returns the codes {00, 01, 10}. Further calls to state-select will search for states with highest column 1-count and therefore, *state4* is selected first. The only code left with the current codelength is {11}, but this code will make the state encoding matrix falsify the constraint relation:

$$A' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad S = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad F = A \cdot S = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ * & 1 \\ * & * \end{bmatrix}$$

$$\bar{F}^2 = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ \phi & \phi \\ 0 & 1 \end{bmatrix} \qquad \bar{F}^2 \wedge F = \begin{bmatrix} 0 & \phi \\ \phi & \phi \\ \phi & \phi \\ 0 & 1 \end{bmatrix} \neq \Phi$$

Therefore, the code space dimension has to be increased and adjoin returns {000, 010, 100}. Again, the candidateset C is empty and now **adjoin** returns {001, 010, 100}. The candidateset is no longer empty and the state encoding matrix becomes

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

As the remaining states al have column 1-count = 1, the order of selecting them is of no importance any more. State *state*3 results in code {011}, *state*2 in code {101} and *state*6 in {110}, as the candidateset is not empty for all these selected states.

It is important to note that the order of the rows of S is not equal to the order of the names of the states given by ESPRESSO, but is dependent on the order in which they are selected to be encoded. Permutating the state encoding matrix according to the order of selecting the states results in the following solution to the constraint state encoding problem for the given FSM:

START	=	010
*state*6	=	110
*state*2	=	101
*state*5	=	100
*state*3	=	011
*state*4	=	000
*state*7	=	001

For this example it is apparent that the state encoding algorithm assigns codes with the *minimal possible codelength* to the *seven* different states. More tests on different FSM's are done by the authors of [1] and reveal that the algorithm itself shows a good performance. The next step will be to make a fast implementation of the algorithm, to be able to do some more tests. The next chapter discusses the implementation in the programming language C.

4. IMPLEMENTATION IN C

In a total system the state encoding algorithm is usually not appreciated as much as it should be. If the algorithm is not used and a random assignment is made, the system will work anyway and the result does not necessarily have to be much worse than the result using the algorithm. So, why use it? On the other hand, if the algorithm is used, it takes quite a lot of computing time and it slows the system down. So, don't use it!

To annul the arguments in reasoning above, two other arguments can be discussed. The whole system of designing integrated circuits has one common factor: *minimization*. Before defining a FSM minimization is performed on external components and after state encoding the combinational component of the FSM is minimized. For the sake of consistency, don't let the state assignment be the bottle-neck, but use an algorithm that minimizes the state register and the combinational component. In a complete system, where some very intelligent algorithms are activated, the total run-time will be much higher than the time it will take an algorithm to encode the states in an appropriate manner. If the result is not amazing, the relative computing time is not either.

The implementation of the algorithm discussed in chapter 3 shall therefore have to satisfy two rules:

- It has to show good performance and
- it must be fast

Concerning the first rule, the authors of [1] prove that the algorithm will minimize the PLA area of the combinational component, using a (sub-)minimal codelength. They illustrate their theorem with some examples of results of the encoding algorithm. The second rule will be satisfied if the right implementation is made. The programming language C is chosen for it, as it can be efficiently compiled, while it still ables the designer to write well structured and easy-to-read programs.

Describing the implementation of the algorithm takes place in three sections. Firstly, the global structure of the program is discussed, showing the resemblance with the structure of the algorithm and defining the global variables used. Secondly, the functions used to build the structure of the program are discussed, omitting evident cases. Finally, the use of the program is discussed, showing the concept of the input and the recognized options. In discussing the implementation, the reader is supposed to have knowledge about the language C. A good manual for it can be found in [2].

4.1 **The global structure of the program**

The structure of the algorithm in section 3.2 already indicates, that the state encoding program uses global variables. Variables like the constraint matrix, the state encoding matrix and the number of states are essentially global. Before the global structure is discussed, the global types and variables will be declared.

4.1.1 *Global types and variables*

Some frequently used types are:

```
int STATENUMBER, BITNUMBER, GROUPNUMBER;
short int BIT, BOOLEAN, EXTENDED_BOOLEAN;
BIT ENCODING_MATRIX[max_nrstates][max_nrbits];
BIT GROUP_MATRIX[max_nrgroups][max_nrstates];
```

For the type BOOLEAN the definitions TRUE = 1 and FALSE = 0 are present. For EXTENDED_BOOLEAN TRUE = 1, FALSE = 0, STAR = 2 and PHI = 3 are defined.

As C does not support the type *set of*, while sets are frequently used in the algorithm, user-defined types are necessary to implement them. Here sets are seen as linked lists of *structures* containing the value of an element and a pointer to the next element (structure). For instance, a set of states is defined as follows:

```
typedef struct state
        {
           STATENUMBER element;
           struct state *next;
        } EL_STATESET, *STATESET;
```

A variable of the type STATESET, e.g. sset, can be visualized as:

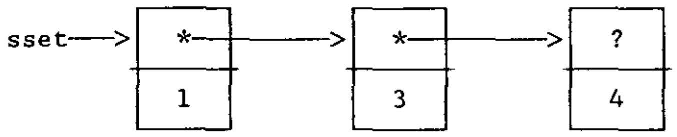

Figure 4.1. Example of a set using a linked list.

The define-statement "define EMPTY NULL" makes the empty set more visual. Adding an element (e.g. 7) to the set means inserting a new structure in front of the list:

```
newptr =  (STATESET) malloc (sizeof(EL_STATESET));
newptr -> element = 7;
newptr -> next = sset;
sset    = newptr;
```

Deleting and membership testing are more complicated, but will not be used in the program. One operation will indeed be frequently used, namely: *for all elements of*:

```
for (walk = sset; walk != EMPTY; walk = walk -> next)
{
   .....
}
```

The elements of a set can also be sets, so even the type *set of set of* may be used. Note that when a variable of the type STATESET has more than one element, a certain order among them is present and will be used in the program. This extension to the type *set of* follows immediately from the implementation with lists.

In the same way the following global types can be defined:

```
CODESET, SET_OF_CODESET;
```

The most frequently used global variables are:

```
STATENUMBER nrstates, nrselected_states, nrencoded_states;
   /* $n_s$, $|\Sigma|$, $|\Gamma|$ */

BITNUMBER nrbits;  /* $n_b$ */

GROUPNUMBER nrgroups;  /* $n_l$ */

GROUP_MATRIX constraint_matrix, current_constr_matrix;
   /* $A$ and $A'$ */
ENCODING_MATRIX state_encoding_matrix;  /* $S$ */
STATESET sel_stateset;  /* $\Sigma$ */
SET_OF_CODESET candidateset;  /* $C$ */
CODESET sel_codeset;  /* $\sigma$ */
```

Using these variables and some functions, the program can be built in a manner that still resembles the structure of the algorithm in section 3.2. There are more global variables than the ones named above. They will be discussed when they appear in the discussion of one of the functions.

4.1.2 *The structure of the main program*

At the group Design Automation (ES), where the implementation of the algorithm took place, the convention exists to give all programs a name preceded by "es", to indicate they were developed there. As the program was called *KISS* in [1], here it will be addressed by the name *eskiss*. KISS stands for Keep Internal States Simple.

The main program is described below. Compare it with the structure of the algorithm in section 3.2:

```
main()
{
   sel_stateset = EMPTY;
   nrencoded_states = 0; nrbits = 0;
   find_primes_of(constraint_matrix);

   while (nrencoded_states < nrstates)
   {
      stateselect(&sel_stateset, &nrselected_states);
      updat_constraint_matrix(sel_stateset);
      candidateset = EMPTY;

      while (candidateset == EMPTY)
      {
         find_candidates(sel_stateset, &candidateset);
         if (candidateset == EMPTY) adjoin(state_encoding_matrix);
         else codeselect(candidateset, &sel_codeset);

      }

      append_to_state_encoding_matrix(sel_codeset);
     /* nrencoded_states = nrencoded_states + nrselected_states; */

   }

}
```

The reason that the statement that updates the variable nrencoded_states is made inactive, is that it is already embedded in the function append_to_state_encoding_matrix. It is written inside comment delimiters to show that the program will terminate, as the while loop control variable will increase.

Comparing the structure of the algorithm and of the main program immediately shows the resemblance and the structure need no more explanation. Discussion of the used functions takes place in section 4.2.

4.1.3 *The file structure*

In C the total program may be divided over different files, making the structure more visual and making it easier to compile the program. Here, each function is stored in a separate file and so are the global declarations and definitions. All global declarations are stored in the file "global_decl.h" and all global definitions in the file "global_def.h". For the difference between declarations and definitions and for the scope of variables, see for instance [2].

At the top of a file containing the source text of a function, the global declarations must be added by means of the statement include "global_decl.h". At the top of the main program both include "global_decl.h" and include "global_def.h" ought to be present. In that way each function has the correct environment and can be written as if it is the only existing function.

4.1.4 *Options*

Some options are built in to make the program more flexible and to add some extensions to the algorithm. The options are passed through the command line arguments and are recognized and processed by a special function (see section 4.2.2). For each option there is a global variable of the type BOOLEAN, indicating whether the specific option was present on the command line or not. In the right functions, in the right place, tests are made upon these option indicators and the corresponding actions are taken. In discussing the separate functions this will become clearer. The different options recognized by eskiss are:

-D	Do not use a dominating set as first selected stateset.
-d n	Allow only n elements in a dominating set.
-e	Suppress all error messages send to stderr.
-r	Generate intermediate result and send them to stderr.
-i NAME	State with name NAME is the initial state and will get code 00...0.

4.2 Discussion of the separate functions

The functions used in eskiss can be divided into two groups:

- globally used functions and
- locally used functions.

The globally used functions are used only in the main program, while the locally used ones are called by the globally used functions. This hierarchy is used to make the discussion of the separate functions easier and this section is divided accordingly.

4.2.1 *Globally used functions*

The structure of the main program was already shown in section 4.1.2, but it was not yet complete, as there are still some functions needed to have it work right. These extra functions were omitted to make the resemblance with the structure of the algorithm more illustrative. The functions are grouped together below and afterwards discussed one by one. How they fit into the main program is evident, but the reader can always check upon it by reading the source text of the program, as for each function the file in which it is stored will be indicated. The main program is stored in the file "eskiss.c".

The globally used functions are:

```
init_defaults()                            "init_def.c"
process_option(option)                     "proc_option.c"
read_input_file_into(A, names)             "read_input.c"
find_primes_of(A)                          "find_primes.c"
stateselect(ss, nrs)                       "stateselect.c"
updat_state_order_matrix(ss)               "updat_matric.c"
updat_constraint_matrix(ss)                "updat_matric.c"
find_candidates(ss, cs)                    "find_cand.c"
adjoin(S)                                  "adjoin.c"
codeselect(cs, scs)                        "codeselect.c"
append_to_state_encoding_matrix(scs)       "append.c"
print_state_encoding()                     "print_codes.c"
```

- **init_defaults()**
  This function initializes the default values for the variables corresponding to the possible options. See also section 4.3.3.

- **process_option(option)**
  The parameter "option" is a string read from the command line. This function checks if it is a legal option and updates the corresponding option indicator (see section 4.3.3). If "option" is not a legal option, an error message is send to stderr.

- **read_input_file_into(A, names)**
  From the standard input file a table generated by the symbolic minimizer ESPRESSO is read and the column of this table corresponding to the one-hot-encoded old-state is entered into *A* (the constraint matrix). As the input to ESPRESSO is a symbolic cover, the input to eskiss will contain one line defining the relation between the statenames and the one-hot-encoding of the states. This relation is stored in the array of strings "names", where names[i] contains the name of the state that is one-hot-encoded as a string of zeros with only one 1 in the ith position from the left. The globals nrstates and nrgroups are also updated. Note that *A* may contain meets as well as primes.

  To read the input file in a structured way, the following locally used functions are called:

- nextsymbol(symbol): collects from the standard input the next symbol and assigns it to "symbol". A symbol is any string of characters, separated by newlines, blanks or tabs.

- checksymbol(symbol, str): checks if "symbol" is equal to the string "str" or not. If not, an error message is send to stderr.

- make_int(symbol, i): assigns to "i" the integer value represented by the string "symbol". If "symbol" contains non-numeric characters an error message is send to stderr,

- equal_symbol(symbol, str): returns TRUE if "symbol" is equal to the string "str", FALSE if not. No error messages are send.

• find_primes_of(A)

After the constraint matrix is filled with the one-hot-encoded old-state, the meets have to be deleted. Remind: A row of *A* is a meet if it is the bit-wise conjunction of two or more state groups. The global nrgoups is updated at the end of the function. Note that according to its definition, the constraint matrix may contain only state groups, i.e. rows with more than one non-zero entry.

Method: Delete all rows with less than two non-zero entries, as those rows cannot be primes. To check if a certain row is a prime or a meet, first find all other rows that have non-zero entries in the same positions as in the current row. Then check if for each zero entry in the current row there exists a row (among the rows with the same non-zero entries) that has a zero entry in the same position. If for each zero entry there exists such a row, the current row is a meet; otherwise it is a prime.

After all rows have been given a label (prime or meet), the ones that are meets are deleted from *A* and nrgroups is updated.

- **stateselect(ss, nrs)**

  A heuristic strategy determines which state or state subset will be selected and assigned to "ss". The number of selected states is returned through the parameter "nrs". The selecting strategy was already explained in section 3.2.

  This function uses one option. At the first invocation of the function, a dominating set is assigned to "ss". If the burden of encoding this dominating set is to great, it is possible to have stateselect assign at the first invocation only one state instead of a dominating set. The option that indicates this is -D. Then the function will be:

```
if (nrencoded_states == 0)  /* First invocation. */

      if ("option -D present")
      {
         find_highest_column_count( &(*ss) );
         *nrs = 1;
      }
      else  /* Use a dominating set. */
         find_dominating_set( &(*ss), &(*nrs) );

else  /* Second or further call. */
{
      find_highest_column_count( &(*ss) );
      *nrs = 1;
}
```

  The locally used functions find_highest_column_count and find_dominating_set search for the state with the highest column 1-count in the constraint matrix and for a dominating set respectively.

- **updat_state_order_matrix(ss)**

  As the states are encoded in an order different from the order of the symbolic names in the input file (see read_input_file_into), each time a stateset ss is selected the order of the elements has to be stored in the matrix state_order_matrix. When state encoding is finished, the state encoding matrix contains the encodings of the states in the order represented by the state_order_matrix. If state_order_matrix[i] = j, then state with number j (position of 1 in one-hot-encoding) has been encoded the ith time and its encoding is the ith row of the state encoding matrix.

- **updat_constraint_matrix(ss)**
  This function adds to the current constraint matrix (A') the columns of the constraint matrix (A) corresponding to the states in the selected stateset ss (Σ).

- **find_candidates(ss, cs)**
  The candidateset cs (C) contains all possible encodings for the selected stateset ss (Σ): $C = \{c$ such that $\begin{bmatrix} S \\ c \end{bmatrix}$ satisfies the constraint relation$\}$.

  Method: Assign all possible codes with length nrbits to the first element of ss. Then find all candidates for the set ss\{first element of ss}. Note that this is recursive.

```
if (ss != EMPTY)

     for (code = 00...0; code <= 11...1; next code)
          if ("code is not yet assigned")
          {
             "assign code";
             find_candidates(ss->next, &(*cs));
          } else;

else  /* All states have a code now. */

     if ("codes satisfy constraint relation")
          "add the codes to the candidateset";
```

  The function tries all possible combinations of codes between 00...0 and 11...1 of length nrbits and checks if the constraint relation is satisfied. If so, the set of codes, corresponding to the selected stateset, is added to the candidateset. Note that most of the time ss contains only one element, as only the first invocation from the main program may have to deal with a dominating set.

The candidateset contains beside the codesets also an indication about the utilization of Boolean space by each specific codeset (see section 3.2). This utilization is determined by the function that checks if the codes satisfy the constraint relation and will be used by the function codeselect, to determine from the candidateset the most suitable codeset. Locally used function: satisfies_constr_rel.

- **adjoin(S)**

  The task of this function was already discussed in section 3.2. Its structure is:

```
if (t < nrselected_states)  return( [S|T] );
else
{
   R' = columns of R not already adjoined to S;
   r  = column of R' with minimal 1-count;

   return( [$\bar{S}$|r] );
}
```

where $T = \{0\}^{n_s \times 1}$, $\bar{S}$ contains the columns of S not equal to T and t is the number of columns of S equal to T. The columns of R are the rows of the current constraint matrix having a non-zero entry in one of the columns corresponding to the states in the selected stateset. Note that when for a certain stateset adjoin is invoked for the first time, t is set to zero, R is updated again and R' = R. Each time it is invoked for the same stateset (candidateset was empty), R remains the same and t or R' is updated.

Now the function becomes:

```
static ... t;
static ... R', R;

if ("not the same stateset as the one just serviced")
{
   t = 0;
   "update R according to the selected stateset";
   R' = R;
}
"determine $\bar{S}$;

if (t < nrselected_states)
{
   t++; nrbits++;
   "add one column of zeros to S";
}
else
{
   r = "column of R' with minimal 1-count";
   "delete column r from R'";
   nrbits = nrbits - t + 1;
   t = 0;
   "add column r to $\bar{S}$";
   S = $\bar{S}$;
}
```

In the real implementation $\bar{S}$ is called *dimE* and a set of already adjoined columns is used instead of R'.

- **codeselect**

  From the candidateset "cs", the best codeset "scs" is selected according to the minimal utilization (see section 3.2 and function find_candidates). Afterwards the candidateset will not be used any more and may therefore be freed using *free(...)*.

```
min_utilization = power2(nrbits);
for ("all elements L of cs")
     if ( L->utilization <= min_utilization )
     {
        min_utilization = L->utilization;
        *scs = L;
     }

"remove the candidateset cs";
```

The locally used function power2(n) returns the nth power of 2 and removing the candidateset is the task of the locally used functions rm_codeset and rm_set_of_codeset.

- **append_to_state_encoding_matrix(scs)**

  The codeset "scs" selected by the previously discussed function has to be added to the state encoding matrix, with the rows in the right order: corresponding to the order of the states in the selected stateset. In "scs" the order of the codes is the other way around than the order of the states in the selected stateset. This situation is ideal for a recursive function:

```
if (scs != EMPTY)
{
   append_to_state_encoding_matrix(scs->next);
   "add first element of scs to state_encoding_matrix";
}
```

  Afterwards the global nrencoded_states is updated and the codeset scs is removed using *free*(...).

- **print_state_encoding()**

  This function sends the result of the state encoding program to stdout. The output consists of "nrstates" lines and each line contains a symbolic statename and its assigned encoding, separated by a tab. The global state_order_matrix is used to determine the right order of the encodings in the output.

  One option is relevant here. Usually, a FSM has one initial state, the state which the system enters after powerup. The most easy way to realize such a powerup state is using a state register with an asynchronous clear input which resets the register to state 00...0. It is therefore useful to be optionally able to define one state as the initial one and make sure that it will get encoding 00...0 assigned.

When state encoding is finished, the constraint relation is invariant under some permutations on the state encoding matrix S. One of these permutations is inversion of all entries in a column. Proof of this theorem follows below. Using this permutation makes it easy to assign 00...0 to the initial state: invert each column of the state encoding matrix having a non-zero entry in the row corresponding to the initial state. This permutation is done by the locally used function ass_initialstate_code_0(E).

Very rarely a column of zero entries only or non-zero entries only may appear in the state encoding matrix. Such a column is called redundant, as it may well be deleted from the matrix: One bit of the state register never changes its value. Within the function print_state_encoding this situation is recognized and removed by the locally used function check_for_redundant_columns.

Theorem:

If S satisfies the constraint relation for A, then $S' = P_i S$ satisfies it too, where the permutation matrix P_i inverts the entries of column i of S.

Proof:

$$\text{Let } S = \left[s_{\cdot 1} \mid s_{\cdot 2} \mid \cdots \mid s_{\cdot i} \mid \cdots \mid s_{\cdot n_b} \right]$$

$$\text{then } F = A \cdot S = \left[f_{\cdot 1} \mid f_{\cdot 2} \mid \cdots \mid f_{\cdot i} \mid \cdots \mid f_{\cdot n_b} \right]$$

$$\text{with } f_{\cdot p} = A \cdot s_{\cdot p}$$

$$\text{and } \bar{F}^k = \left[\bar{f}^k_{\cdot 1} \mid \cdots \mid \bar{f}^k_{\cdot i} \mid \cdots \mid \bar{f}^k_{\cdot n_b} \right]$$

with $\bar{f}^{k}_{\cdot p} = \left(\bar{a}_{\cdot k} \cdot s_{k\cdot} \right)_{\cdot p} = \bar{a}_{\cdot k} \cdot s_{kp}$

and $F \wedge \bar{F}^{k} = \left[\begin{array}{c|c|c|c|c} f_{\cdot 1} \wedge \bar{f}^{k}_{\cdot 1} & \cdots & f_{\cdot i} \wedge \bar{f}^{k}_{\cdot i} & \cdots & f_{\cdot n_b} \wedge \bar{f}^{k}_{\cdot n_b} \end{array} \right] = \Phi$

for $k = 1, 2, \cdots, n_l$

Now $S' = \left[\begin{array}{c|c|c|c|c|c} s_{\cdot 1} & s_{\cdot 2} & \cdots & \bar{s}_{\cdot i} & \cdots & s_{\cdot n_b} \end{array} \right]$

Let the inversion of ϕ and $*$ be defined as ϕ and $*$ respectively.

Then $F' = A \cdot S' = \left[\begin{array}{c|c|c|c|c|c} f_{\cdot 1} & f_{\cdot 2} & \cdots & \bar{f}_{\cdot i} & \cdots & f_{\cdot n_b} \end{array} \right]$

and $(\bar{F}^{k})' = \left[\begin{array}{c|c|c|c|c} \bar{f}^{k}_{\cdot 1} & \cdots & \bar{x}^{k}_{\cdot i} & \cdots & \bar{f}^{k}_{\cdot n_b} \end{array} \right]$

with $\bar{x}^{k}_{\cdot p} = \left(\bar{a}_{\cdot k} \cdot s_{k\cdot}' \right)_{\cdot p} = \bar{a}_{\cdot k} \cdot s_{kp}' = \bar{a}_{\cdot k} \cdot \bar{s}_{kp}$

so $F' \wedge (\bar{F}^{k})' = \left[\begin{array}{c|c|c|c|c} f_{\cdot 1} \wedge \bar{f}^{k}_{\cdot 1} & \cdots & \bar{f}_{\cdot i} \wedge \bar{a}_{\cdot k} \cdot \bar{s}_{ki} & \cdots & f_{\cdot n_b} \wedge \bar{f}^{k}_{\cdot n_b} \end{array} \right] = \Phi$

for $k = 1, 2, \cdots, n_l$

The only difference between $F \wedge \bar{F}^{k}$ and $F' \wedge (\bar{F}^{k})'$ is the ith column. Consider now the jth element of those columns:

$$\begin{cases} \left(F \wedge \bar{F}^{k} \right)_{ji} = f_{ji} \wedge (\bar{a}_{jk} \cdot s_{ki}) \\ \left(F' \wedge (\bar{F}^{k})' \right)_{ji} = \bar{f}_{ji} \wedge (\bar{a}_{jk} \cdot \bar{s}_{ki}) \qquad 1 \le j \le n_l,\ 1 \le i \le n_b \end{cases}$$

Then there are two possibilities:

1. $\bar{a}_{jk} = 0$ results into:

$$\begin{cases} f_{ji} \\ \bar{f}_{ji} \end{cases}$$

Remember that $F \wedge \bar{F}^k = \Phi$!!

- If $f_{ji} \in \{0,1,*\}$ then row j of $F \wedge \bar{F}^k$ contains at least one ϕ entry on position(s) not equal to i. As the entries of row j of $F' \wedge (\bar{F}^k)'$ on those positions are equal to the same entries in $F \wedge \bar{F}^k$, it follows that $F' \wedge (\bar{F}^k)' = \Phi$ for $k = 1, 2, \ldots, n_1$ and S' satisfies the constraint relation.

•• If $f_{ji} = \phi$ then $\bar{f}_{ji} = \phi$ and

$F' \wedge (\bar{F}^k)' = \Phi$ for $k = 1, 2, \ldots, n_1$ and S' satisfies the constraint relation.

Since there are no other possibilities for f_{ji}, it is proved that for $\bar{a}_{jk} = 0$ S' satisfies the constraint relation.

2. $\bar{a}_{jk} = 1$ results into:

$$\begin{cases} f_{ji} \wedge s_{ki} \\ \bar{f}_{ji} \wedge \bar{s}_{ki} \end{cases}$$

••• If $f_{ji} = s_{ki} \in \{0,1\}$ or $f_{ji} = *$ then $f_{ji} \wedge s_{ki} = \overline{\bar{f}_{ji} \wedge \bar{s}_{ki}} \in \{0,1\}$ and the same reasoning as after • can be given to prove that S' satisfies the constraint relation.

•••• If $f_{ji} = \bar{s}_{ki} \in \{0,1\}$ or $f_{ji} = \phi$ then

$f_{ji} \wedge s_{ki} = \bar{f}_{ji} \wedge \bar{s}_{ki} = \phi$ and

$F' \wedge (\bar{F}^{k})' = \Phi$ for $k = 1, 2, \ldots, n_1$ and S' satisfies the constraint relation.

Since there are no other possible combinations for f_{ji} and s_{ki}, it is proved that for $a_{jk} = 1$ S' satisfies the constraint relation.

From the two possibilities 1. and 2. for $\bar{a}_{jk}$ it follows that in general S' satisfies the constraint relation if S does.

4.2.2 *Locally used functions*

Functions that are not directly called from the main program, but are used in other functions, are called locally used functions. In the previous subsection some of them were already encountered, here they are all discussed in order. The names and the files follow:

nextsymbol(symbol)	"nextsymbol.c"
checksymbol(sl, s2)	"checksymbol.c"
equalsymbol(sl, s2)	"equalsymbol.c"
make_int(s, i)	"make_int.c"
find_dominating_set(dom_set, card)	"find_domset.c"
make_first_list(stptr)	"make_tree.c"
make_search_tree(stptr, dom_set)	"make_tree.c"
longest_path(stptr, dom_set, length)	"longest_path.c"
remove_search_tree(stptr)	"remove_tree.c"
find_highest_column_count(ae, A, ss)	"find_high_cc.c"
satisfies_constr_rel(cs, ut)	"sat_con_rel.c"
disjunction(bl, b2)	"condisjunct.c"
conjunction(bl, b2)	"condisjunct.c"
check_for_redundant_columns(S, R)	"check_redun.c"
ass_initialstate_code_0(E)	"ass_init.c"
convert_to_binary(i, c)	"conv_to_bin.c"
power2(i)	"power2.c"
element_of_set(c, sc)	"element_of.c"
element_of_array(c, A)	"element_of.c"

- **nextsymbol(symbol)**
  Collects from the standard input the next symbol.

- **checksymbol(s1, s2)**
  If string s1 is not equal to string s2 an error message is send to stderr.

- **equalsymbol(s1, s2)**
  Returns TRUE if string s1 is equal to string s2, FALSE otherwise. No error messages are generated.

- **make_int(s, i)**
  The parameter i becomes equal to the integer value represented by the string s. If s contains non-numeric characters an error message is send to stderr.

- **find_dominating_set(dom_set, card)**
  This function searches in the constraint matrix for a dominating set and assigns that set to dom_set and its cardinality to card. The problem of finding a dominating set is very complex and must be solved through intelligent searching. One possibility is using a *search-tree*, that basically has the following form:

  Each element of the search-tree contains a statenumber and two pointers. The statenumber corresponds to a column of the constraint matrix. The first pointer (sub_tree) points to a list of "sons", which forms the basis of a sub search-tree. Elements chained together with the second pointer (rest_list) are all "brothers" of each other. Brothers of an element that can be reached by walking through pointers rest_list in the correct direction, are called "younger brothers". "Elder brothers" can only be reached by walking back, against the direction of the pointers.

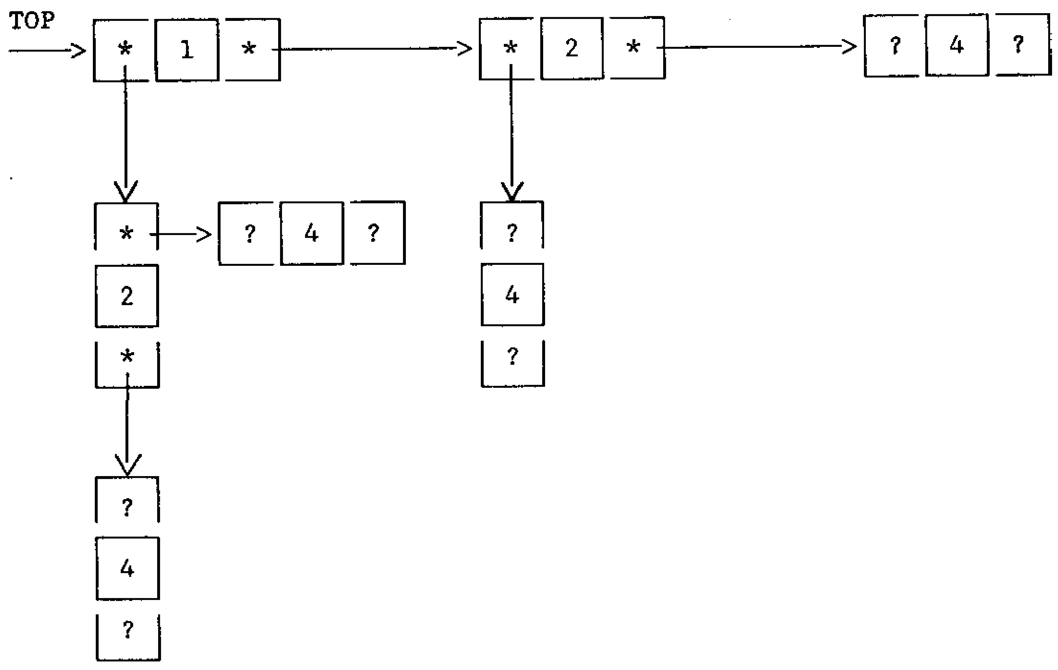

Figure 4.2. Basic form of a search-tree.

Rule 1: Each list of brothers hanging under element *Y*, through pointer *Y.sub_tree*, contains all elements that – each individually – may be part of a dominating set, in which also Y is present.

Rule 2: The set of sons of an element is a subset of the set of younger brothers of that element.

Theorem: If a search-tree is constructed satisfying the above rules, then the longest path possible, starting from the begin pointer TOP and walking through pointers sub_tree (possibly skipping over elements through pointers rest_list), gives the dominating set.

Using this theorem, finding a dominating set can be realized as follows:

```
find_dominating_set(dom_set, card)
{
   make_first_list(&TOP);
   make_search_tree(TOP);
   longest_path(TOP, &(*dom_set), &(*card));
   remove_search_tree(TOP);

}
```

Firstly, the list of brothers consisting of all states is created and it satisfies rule 1, while rule 2 is still irrelevant. Then the remainder of the search-tree is created, such that rule 1 and rule 2 are never violated. According to the theorem above, the dominating set can be found by searching for the longest path. Afterwards, the search-tree will not be used any more and may be removed using *free*(...).

- make_first_list(stptr)

  One new type is introduced:

```
struct search_tree_el
{
   STATENUMBER value;
   struct search_tree_el *sub_tree, *rest_list;
} EL_SEARCH_TREE, *SEARCH_TREE_PTR;
```

  Further details are evident.

- make_search_tree(stptr, dom_set)

  (The extra parameter "dom_set" is necessary as this function uses recursion.) Formally, this function creates the sub search-tree with begin pointer stptr, satisfying rule 1 and rule 2 and the restriction that all elements of the resulting sub search-tree may – each individually – be part of a dominating set containing also the set of states dom_set.

  Stated in this way, the function can easily be described using recursion:

```
make_search_tree(stptr, dom_set)
{
   for ("all elements L of list stptr->...")
   {
      "add L->value to dom_set";
      for ("all elements B of list L->rest_list->...")
      {
         if ("B->value may be part of a dominating set
              containing also dom_set")
         {
            "add an element with value B->value to the
             list of sons under L->sub_tree";
         }
      }
      make_search_tree(L->sub_tree, &(*dom_set));
      "remove L->value from dom_set";

   }

}
```

It is easy to see, that rule 1 and rule 2 are never violated, while the same is true for the restriction that each element can be part of a dominating set containing also dom_set.

One option is relevant here. The maximal cardinality of a dominating set may be initialized by the user. This means, that the user may restrict the number of states in a dominating set to a certain value "max_carddomset". When the complete dominating set would exceed this value, a semi-dominating set, containing exactly max_carddomset elements, will be returned. The set is called a semi-dominating set as the complete dominating set has by definition maximal cardinality, while the semi-dominating set has less elements. The option is given as: -d n, where n is the defined maximal cardinality. If this option is absent, max_carddomset is set to 3.

Implementing this option requires statements that abort the further construction of the search-tree as soon as the cardinality is equal to max_carddomset. Then, longest_path will always find a semi-dominating set with cardinality max_carddomset. The function make_search_tree will return TRUE if max_carddomset is not exceeded, FALSE otherwise. This ables the function find_dominating_set to send an intermediate result to stderr if exceeding max_carddomset occurs and if the option -r is present.

- **longest_path(stptr, dom_set, length)**
  This recursive function assigns to "dom_set" a pointer to the set of states that belong to the dominating set. The cardinality of the set is returned through "length" and the begin pointer of the (sub) search-tree is "stptr". The function then becomes:

```
longest_path(stptr, dom_set, length)
{
   SEARCH_TREE_PTR longest;
   STATESET d;
   STATENUMBER len;

/* Firstly, determine from the brothers in the list stptr->...
   the one that has the longest path under it through sub_tree
*/

   *length = 0; longest = stptr; *dom_set = EMPTY;
   for ("all brothers B of stptr->...")
   {
      longest_path(B->sub_tree, &d, &len);
      if (len > *length)
      {
         *length  = len;
         *dom_set = d;
         longest  = B;
      }

   }

/* Now *length is the length of the path *dom_set under brother
   longest*
*/

   if (longest != EMPTY)
        "add this element to *dom_set";

}
```

- **remove_search_tree(stptr)**

  When the dominating set is found, the search-tree is not needed any more. Removing this tree is realized as follows:

  ```
  for ("all brothers B of stptr->...")
       remove_search_tree(B->sub_tree);

  "remove all brothers B";
  ```

- **find_highest_column_count(ae, A, ss)**

  The array "ae" stands for "already encoded" and indicates which states are already encoded (ae[.] = TRUE) and which are not (ae[.] = FALSE). Then, this function assigns to "ss" an element of a stateset with value the *first state that* has the highest column 1-count in the constraint matrix "A" among the states that have ae[.] = FALSE. Method: simply count the non-zero entries in each column with ae[.] = FALSE and select the highest result.

- **satisfies_constr_rel(cs, ut)**

  To determine whether the current state encoding matrix, augmented by the codeset "cs", satisfies the constraint relation for the current constraint matrix, the matrices $F = A \cdot S$ and $\bar{F}^i = \bar{a}_{\cdot i} \cdot s_{i\cdot}$ have to be determined. This is straightforward (see also section 2.3). Then, the relation $F \wedge \bar{F}^i = \Phi$ has to be verified and if the relation is satisfied for all i, the function returns TRUE, otherwise FALSE. The matrices F and $\bar{F}^i$ are called *face_matrix* and *dim_inv_face_matrix* respectively.

  During the construction of F the *utilization* of Boolean space, "ut", can be determined, according to (see section 3.2):

  $$ut = \min_{\text{all faces}} (2^{\text{number of } *}) \qquad \text{with } * \text{ is the don't care value}$$

where a *face* is a row of the *face_matrix*. The value of ut is used by the globally used function codeselect.

- **disjunction(b1, b2)**
  Returns the disjunction of the EXTENDED_BOOLEAN variables "b1" and "b2". See section 2.3.

- **conjunction(b1, b2)**
  Returns the conjunction of the EXTENDED_BOOLEAN variables "b1" and "b2". See section 2.3.

- **check_for_redundant_columns(S, R)**
  A redundant column is one with zero entries only or non-zero entries only. For all i, this function assigns to "R[i]" TRUE if column i of state encoding matrix "S" is redundant, FALSE otherwise.

- **ass_initialstate_code_0(E)**
  If the option -i NAME is present on the command line, the state with name NAME must be assigned code 00...0. This function simply inverts all columns of the state encoding matrix "E" that have a non-zero entry in the row corresponding to the initial state. The number of the initial state is determined by the function read_input_file_into and is assigned to the global variable "initial_state". Inversion of an entire column leaves the constraint relation invariant, as proved by the theorem in the discussion of the globally used function print_state_encoding.

- **convert_to_binary(i, c)**
  The function find_candidates needs the statement

```
for (code = 00...0; code <= 11...1; next code)
{
   .....

}
```

This is realized by means of this function as follows:

```
     for (c = 0; c <= (power2(nrbits)-1); c++)
     {
        convert_to_binary(c, sel_code);
        .....

     }
```

(The description of power2(i) follows below.) This function converts the integer i into a binary code with length "nrbits", preceded by zeros if necessary. Method:

```
     k = 0;
     while (i > 0)
     {
        k++;
        c[k] = i % 2;   /* i MOD 2 */
           i = i / 2;   /* i DIV 2 */

     }

     "add necessary leading zeros";
```

- **power2(i)**
  Returns the ith power of 2, where "i" must be a positive integer.

- **element_of_set(c, sc)**
  Checks whether the code "c" is an element of the codeset "sc" and returns TRUE if so, FALSE otherwise. The length of the code must be equal to the global "nrbits".

- **element_of_array(c, A)**
  Checks whether the code "c" with length "nrbits" is contained within the ENCODING_MATRIX "A". If so, the function returns TRUE, otherwise FALSE. Note that A has "nrencoded_states" rows and "nrbits" columns.

4.3 The concept of the input

It was mentioned earlier, that ESPRESSO takes as input a different format than the one discussed in section 2.2. It is therefore needed to preprocess the input before sending it to ESPRESSO. After describing the input to ESPRESSO and the input to the preprocessor (or *formator*), the latter program is discussed, followed by the construction of the formator and eskiss into a complete system for symbolic input state assignment.

4.3.1 *ESPRESSO: input and output*

Appendices 1 and 2 show the format definitions of the input and output concept for ESPRESSO respectively. The input concept differs slightly from the input concept for the complete system. The most important differences are (see also appendix 3: input concept for FORMAT):

- The order of the variables (columns) in the transition table;
- symbolic values for the output variable.

Here, the need for a preprocesser becomes clear. It would be very inconsistent if the output variable could not be given using symbolic values. The preprocessor *does* allow the use of symbolic values.

The output generated by ESPRESSO is straightforward: the heading contains the global "nrstates" and the label-list contains the names of the states. The second to last column forms the expanded constraint matrix, where meets as well as primes may occur. This concept is immediately accepted by eskiss.

4.3.2 *The preprocessor FORMAT*

The output concept for the preprocessor is of course the same as the input concept for ESPRESSO. The differences between the input concepts for ESPRESSO and for the preprocessor have been discussed in the previous subsection and the translation of the latter into the former is straightforward. The description of the input concept for the preprocessor can be found in appendix 3.

The program that performs the translation is called *format* and uses stdin and stdout as input and output files respectively. Format uses the functions nextsymbol and make_int that have already been discussed in the previous section. It also uses one new function called one_hot_encode(val, size). This function encodes the symbolic value given by the parameter "val", using a one-hot-encoding scheme with length equal to "size". At the *i*th time the function is invoked, it returns a string of "size" zeros, except for one 1 in position *i*. The function is necessary as ESPRESSO needs an output variable that is one-hot-encoded rather than symbolic.

The source text of the preprocessor is stored in file "format.c" and the function one_hot_encode in file "one_hot_enc.c".

4.3.3 *Using eskiss, options*

Input passed through stdin, formated as given by appendix 3, may be entered into the following system, resulting in a state encoding program, sending its output to stdout:

```
format | espresso | eskiss  [options]
```

Input may come from the terminal, from an existing file using cat... or from another program using a pipeline "|". Output may go to the terminal screen, to a file using redirection ">" or to another program using "|".

The options are passed to eskiss as command line arguments. Each option starts with a minus sign "-" and is separated from other options by a blank. Two options may be appended to one single minus sign, if the first option consists of a single character, otherwise they must be separated. So -ri START -e is correct, while -d 2r -e is incorrect, as -d 2 is one option, consisting of more than one character.

The recognized options are:

-D	Do not use a dominating set when selecting the first stateset, but select a state with highest column count.
-d n	The maximal cardinality of a dominating set is set to n. If a dominating set would exceed this maximal cardinality, only the first n elements of that set will be selected, instead of the complete dominating set.
-r	Generate intermediate results and send them to stderr. A list of all intermediate result messages can be found in appendix 4.
-e	Suppress all error messages to stderr. Note that intermediate results are not suppressed! A list of all error messages can be found in appendix 5.
-i NAME	The state with name NAME is taken to be the initial state and will get code 00...0 assigned to it. If NAME is not a legal name, an error message is send to stderr and no initial state is assumed. Any other (legal) name may be used.

5. EXAMPLES

To be able to say something about the performance of the resulting program, some tests on examples of FSM's have to be done. It is of course necessary to use examples that correspond to realistic systems, rather than to just some random state transitions, as in practice only realistic systems will be entered. In [1] some results of the program KISS are given, based on examples that are also available at the group Automatic System Design, at the Eindhoven University of Technology. They are stored in the files ass*X*.mv, where *X* stands for the number of the example, 1 ≤ *X* ≤ 7.

The table below shows an excerpt of the results obtained by KISS and by eskiss.

				KISS		eskiss	
FSM	ni	no	ns	nb	time(sec)	nb	time(sec)
FSM 1	4	1	5	3	4	3	1.6
FSM 2	8	5	7	5	31	5	14.1
FSM 3	8	5	4	4	10	4	6.9
FSM 4	4	3	27	9	748	?	? *
FSM 5	4	3	8	4	11	5	7.4 **
FSM 6	2	2	7	3	4	3	1.3
FSM 7	2	3	15	5	26	5	11.0

ni = number of input symbols
no = number of output symbols
ns = number of internal states
nb = number of bits per code

* and ** : See following text.

Comparing the results of the different programs, it is clear that eskiss shows about the same or even better performance as KISS does, when FSM's with little number of states (15) are concerned. FSM 5 is implemented by eskiss using one bit more than the implementation generated by KISS. Even if a full dominating set is used to select the first stateset, a 5-bit solution is found. The cause of this difference can be the fact that a dominating set is not necessarily unique. If KISS finds an other dominating set than eskiss does, a different result may appear.

KISS generates a 9-bit codelength solution within about 12 minutes for FSM 4. For the same FSM eskiss does not give any result. Using the full dominating set takes too much memory to store the search-tree (section 4.2.2) and the program terminates abnormally. If the full dominating set is not used, the resulting codelength will be longer than the resulting 9 bits of KISS. Reaching a codelength of more than 12 bits makes the program run for hours.

The main reason that the program becomes that time-consuming when greater codelengths are concerned, is the bad implementation of the function find_candidates. Using the option -r on the command line (sending intermediate results to stderr) shows indeed, that the program runs properly, but needs a lot of time to find the set of candidates. The effect of this bad implementation becomes apparent as soon as a codelength of about 12 bits has been obtained.

Calling the program 'useful' is allowed only after changing the implementation of this function, to make it run much faster. More about this topic can be found in the next chapter.

6. CONCLUSIONS

After testing the resulting program for optimal state assignment in the previous chapter, some conclusions and suggestions for improvement can be stated. As the program is not yet implemented as efficient as it should and could be, a task is still lying open for a practical traineeship. An indication to what direction has to be taken to improve the program, is given below. Thus, this report and the documentation inside the program source text are sufficient literature for other investigators.

Chapter 5 already indicated that the function find_candidates has to be implemented more efficiently, as this is the most time-consuming part of the entire program. For FSM's with a small number of internal states, the effect of the inefficiency of this function stays unnoticed. As soon as the number of states increases, the effect is exponentially increased and the program is not useful any more.

The main reason for find_candidates to be that time-consuming, is the fact that it tests the constraint relation for each possible code with the current codelength. So, if the current length is n, the function satisfies_constr_rel is called 2^n times at one invocation of find_candidates. Implementing this function more efficiently will therefore have the greatest effect on total process time.

The first improvement of the function satisfies_constr_rel may come from the more efficient implementation of the different matrices like the constraint matrix and the state encoding matrix. As they are now implemented as two-dimensional arrays of integers, efficiency is very low. A better way would be the implementation using bit operations, as allowed in the programming language C. This may speed up the calculation of the intermediately used matrices inside satisfies_constr_rel.

A second improvement can be effectuated by updating the intermediately used matrix instead of calculating them fully, each time they are needed. Research on the possibilities of this implementation have to be done first.

And last but certainly not least, an improvement can be realized using the fact that the utilization of Boolean space is calculated before the complete test of the constraint relation is executed. As out of the set of candidates only one code is selected according to its utilization (see chapter 4), only those candidates that show an optimal utilization need to be substituted into the constraint relation. In that way it is possible, to have satisfies_constr_rel execute not only the testing of the constraint relation, but also the selecting of the best candidate according to the utilization. Thus, find_candidates will return only one candidate, the one that has the most optimal utilization. This saves a great deal of computing time, as the constraint relation is tested many times less than it is in the current version of eskiss.

Improving the function find_candidates itself can also be realized. Now, all codes with the current codelength are regarded. Suppose the function would stop testing after it has found a certain number of candidates. Then, only a fraction of the process time will be used, while the optimal solution may still be part of the found set of candidates.

A completely different approach to the method for finding the set of candidates can be regarded. Suppose a constructive function is used instead of a searching one. The construction of the set of candidates can probably be done in a fraction of the time needed to search for the same set. Construction should be executed based on the constraint relation, using the current constraint matrix and the state encoding matrix. This topic will probably be the most effective one, as it changes the order of the method to find the candidateset, while all other suggestions introduce only marginal changes. Research on this topic will be necessary and will probably result in an effectively implemented function.

As mentioned above, the suggestions for improvement of the program leave a task for a practical traineeship. Implementation of some of the suggestions above may result into a program for optimal state assignment, working correctly and within reasonable time.

REFERENCES

[1] De Micheli, G. and R.K. Brayton, A. Sangiovanni-Vincentelli
OPTIMAL STATE ASSIGNMENT FOR FINITE STATE MACHINES.
IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., Vol. CAD-4(1985), p. 269-285.

[2] Kernighan, B.W. and D.M. Ritchie
THE C PROGRAMMING LANGUAGE.
Englewood Cliffs, N.J.: Prentice-Hall, 1978.
Prentice-Hall software series

[3] Hezewijk, J.G. van
TRANSLATION OF A SET OF REGULAR EXPRESSIONS INTO A FINITE STATE MACHINE.
Practical Work Report. Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology, 1987. (In preparation).

[4] McNaughton, R. and H. Yamada
REGULAR EXPRESSIONS AND STATE GRAPHS FOR AUTOMATA.
IRE Trans. Electron. Comput., Vol. EC-9(1960), p. 39-47.

[5] Floyd, R.W. and J.D. Ullman
THE COMPILATION OF REGULAR EXPRESSIONS INTO INTEGRATED CIRCUITS.
J. Assoc. Comput. Mach., Vol. 29(1982), p. 603-622.

[6] Hill, F.J. and G.R. Peterson
INTRODUCTION TO SWITCHING THEORY AND LOGICAL DESIGN. 2nd ed.
New York: Wiley, 1974.

[7] Stok, L. and R. van den Born, G.L.J.M. Janssen
HIGHER LEVELS OF A SILICON COMPILER.
Department of Electrical Engineering, Eindhoven University of Technology, 1986.
EUT Report 86-E-163.

APPENDIX 1: INPUT CONCEPT FOR ESPRESSO

```
(*
Definition of input to ESPRESSO.
*)


<ESPRESSO-input> ::= <heading> <body> <end> .

<heading> ::= ".mv" <nrvars> <nrbinary-vars>
              { <sizeof-mv-input> }
                <sizeof-old-state> <sizeof-new-state>
                <sizeof-mv-output>
              "<RETURN>"
              ".kiss" .

<nrvars> ::= "positive-integer" .

<nrbinary-vars> ::= "0" .  (* Only multiple valued variables used. *)

<sizeof-mv-input> ::= "negative-integer" .

<sizeof-old-state> ::= "negative-integer" .

<sizeof-new-state> ::= "negative-integer" .

<sizeof-mv-output> ::= "positive-integer" .  (* Output not symbolic *)

<body> ::= { <implicant> }+ .

<implicant> ::= { <input> } <old_state> <new-state> <output> .

<input> ::= "symbolic-name" .

<old-state> ::= "symbolic-name" .

<new-state> ::= "symbolic-name" .

<output> ::= "one-hot-encoded-name" .

<end> ::= ".e" .
```

(*
Symbols may be separated by white-space characters (TABs, newlines, blanks). A "symbolic-name" is any sequence of two or more characters, EXCEPT FOR THE CHARACTER "|". THE NAMES ".mv", ".label", ".kiss", ".e" ".end" AND ".p" ARE ALSO NOT ALLOWED.

The number of occurrences of <sizeof-mv-input> must be equal to: <nrvars> - 3. The number of different symbolic names must for all variables but the output, be equal to their corresponding size in the header: <sizeof-mv_input>, <sizeof-old-state> and <sizeof-new-state>. The same applies to the one-hot-encoded values of the output variable and <sizeof-mv-output>. Note that this is not necessarily true for the new-state. The number of <input> in <implicant> must be equal to: <nrvars> - 3.
*)

APPENDIX 2: OUTPUT CONCEPT FOR ESPRESSO

```
(*
Definition of output format from ESPRESSO,  when the input was format-
ted according to appendix 1.
*)


<ESPRESSO-output> ::= <heading> <label-list> <nr-implicants>
                      <body> <end> .

<heading> ::= ".mv" <nrvars> <nrbinary-vars> { <sizeof-mv-input> }
              <nrstates> <sizeof-combined-output> .

<nrvars> ::= "positive-integer" .

<nrbinary-vars> ::= "0" .  (* Only multiple valued variables used. *)

<sizeof-mv-input> ::= "positive-integer" .

<nrstates> ::= "positive-integer" .

<sizeof-combined-output> ::= "positive-integer" .

<label-list> ::= { <label> }+ .

<label> ::= ".label" "var=" <var-number> { <value-name> }+ .

<var-number> ::= "non-negative-integer" .

<value-name> ::= "symbolic-name" .

<nr-implicants> ::= ".p" "positive-integer" .

<body> ::= { <implicant> }+ .

<implicant> ::= { <input> } <old-state> <combined-output> .

<input> ::= "one-hot-encoded-name" .

<old-state> ::= "one-hot-encoded-name" .

<combined-output> ::= "one-hot-encoded-name" .

<end> ::= ".e" .
```

(*
Symbols may be separated by white-space characters (TABs, newlines, blanks). A "symbolic-name" is any sequence of two or more characters, EXCEPT FOR THE CHARACTER "|". THE NAMES ".mv", ".label", ".kiss", ".e" ".end" AND ".p" ARE ALSO NOT ALLOWED.

The number of occurrences of <sizeof-mv-input> will be equal to: <nrvars> - 2. The number of <label> in <label-list> is equal to: <nrvars> - 1. For each <label> the number of <value-name> is equal to the corresponding <sizeof-mv-input> or <nrstates> in <heading>. The number of <input> in <implicant> is equal to <nrvars> - 2.

The last line of <label-list> contains the definition of the names of the states corresponding to the one-hot-encoded old-state.

The <combined-output> is the combination of the one-hot-encoded old-state and the output variable.
*)

APPENDIX 3: INPUT CONCEPT FOR FORMAT

```
(*
Definition of input to state encoding program:
FSMDL, Finite State Machine Description Language.
*)


<FSMDL> ::= <heading> <body> <end> .

<heading> ::= <nrinputs> { <sizeof-input> }
              <nroutputs> { <sizeof-output> }+
              <nrstates> .

<nrinputs> ::= "positive-integer" .

<sizeof-input> ::= "positive-integer" .

<nroutputs> ::= "positive-integer" .

<sizeof-output> ::= "positive-integer" .

<nrstates> ::= "positive-integer" .

<body> ::= { <implicant> }+ .

<implicant> ::= { <input> } { <output> }+ <old-state> <new-state> .

<input> ::= "symbolic-name" .

<output> ::= "symbolic-name" .

<old-state> ::= "symbolic-name" .

<new-state> ::= "symbolic-name" .

<end> ::= ".e" .
```

(*
Symbols may be separated by white-space characters (TABs, newlines, blanks). A "symbolic-name" is any sequence of two or more characters, EXCEPT FOR THE CHARACTER "|". THE NAMES ".mv", ".label", ".kiss", ".e" ".end" AND ".p" ARE ALSO NOT ALLOWED.

The number of occurrences of <sizeof-input> and <sizeof-output> must be equal to <nrinputs> and <nroutputs> respectively. The number of different symbolic names must for all variables (inputs, outputs, old-state, new-state) be equal to their corresponding size in the header: <sizeof-input>, <sizeof-output>, <nrstates>. Note that this is not necessarily true for the new-state. The number of <input> and <output> in <implicant> must be equal to <nrinputs> and <nroutputs> respectively.
*)

APPENDIX 4: LIST OF INTERMEDIATE RESULTS

The following messages will be send to stderr if the option -r is given on the command line. The order in which they will be send depends on the progress of the state encoding program.

```
"reading standard input"
"searching for primes"
"constraint matrix is:"
"selecting first stateset"
"searching for dominating set"
"maximum cardinality of dominating set exceeded"
"continue with cardinality is .."
"cardinality dominating set is .."
"selected states are: ..."
"selecting next state"
"searching for state with highest column count"
"selected state is: ..."
"searching for candidates"
"candidateset is empty"
"selecting code(s)"
"state encoding matrix is:"
"adjoining"
"Now state encoding matrix is:"
"assign to state .. code 00...0"
"state encoding finished"
```

APPENDIX 5: LIST OF ERROR MESSAGES

The following error messages can be send to stderr if run-time errors occur. If the option -e is present on the command line, error messages are suppressed.

```
"ERROR: options must start with '-' "
"unknown option: ..."
"option will be ignored"
"ERROR: the given initial state name is illegal"
"The program will continue without initial state"
"ERROR: no binary variables allowed."
"Use two-valued MV variables."
"WARNING: Characters after '.e' "
"         Remaining characters are ignored and program
          continues"
"ERROR: ... expected"    where ... is a string of characters
"ERROR: integer expected"
```

APPENDIX 6: MANUAL INPUT USING REGULAR EXPRESSIONS

It is sometimes desirable to be able to enter the description of a FSM by hand and to have the program assign the optimal encoding to the states. This approach requires an input language that makes the input easy to enter and easy to read by the user. On this subject a student is working and results are not yet known. Referring to [3] means waiting for the report of traineeship by that student.

The idea behind the input language is formed by [4] and [5], where the concept of *regular expressions* is discussed and where an algorithm is presented to translate a description using regular expressions into a state-diagram. Once a state-diagram for a FSM is known, the transition table is easily constructed and the input formated according to appendix 3 can be send to the state encoding system. Some extensions to the algorithm have to be made, as the original algorithm can handle only regular expressions indicating a FSM with only one input and one output. More about this can be found in [3].

Regular expressions able the user to design his FSM on a higher level than ordinary states. A regular expression indicates what happens if a certain order of input signals occurs and says nothing about how this should be implemented using states. Defining states is the task of the translating algorithm. More about these subjects can be found in [4] and [5].

SUPPLEMENT

As mentioned in chapter 6, the most promising way to alter the function find_candidates in order to speed up the program for state encoding, is to have the function *construct* the set of candidates instead of search for it. The construction will be based on some formulae stated below.

Suppose the state encoding matrix S' with b columns and m rows satisfies the constraint relation for the current constraint matrix A', having m columns and n rows. Constructing the candidateset is the problem of determining all possible codes σ with length b, so that S satisfies the constraint relation for A, with:

$$A = [\ \underset{m}{A'}\ |\ \underset{1}{a}\ ]\ n \qquad \text{and} \qquad S = \underset{b}{\begin{bmatrix} S' \\ \sigma \end{bmatrix}} \begin{matrix} m \\ 1 \end{matrix}$$

where a is the column of the overall constraint matrix corresponding to the state yet to be encoded.

Now $F' = A' \cdot S'$ and $(\bar{F}^i)' = \bar{a}'_{\cdot i} \cdot s'_{i\cdot}$ and for all i $\bar{F}' \wedge (\bar{F}^i)' = (\Phi^i)' = \Phi$, where Φ has at least one ϕ entry on each row.

It is easy to see, that

$$F = A \cdot S = F' \vee (a \cdot \sigma)$$

$$\bar{F}^i = \begin{cases} (\bar{F}^i)' & \text{for } i \leq m \\ \bar{a} \cdot \sigma & \text{for } i = m+1 \end{cases}$$

Firstly, consider for $i \leq m$ the k^{th} row of $F \wedge \overline{F}^i$:

$$(F \wedge \overline{F}^i)_{k\cdot} = (F' \wedge (\overline{F}^i)')_{k\cdot} \ \vee \ ((a \cdot \sigma)_{k\cdot} \wedge (\overline{F}^i)'_{k\cdot}) =$$

$$= (\Phi^i)'_{k\cdot} \ \vee \ ((a_k \cdot \sigma) \wedge (\overline{F}^i)'_{k\cdot})$$

For each k there is at least one p, such that $(\Phi^i)'_{kp} = \phi$. So, if $(\Phi^i)'_{kp} = \phi$ and S satisfies the constraint relation for A, then the following relation must be true for all k:

$$(a_k \cdot \sigma_p) \wedge (\overline{F}^i)'_{kp} = \phi$$

Thus, if $a_k = 0$ then $\sigma_p \in \{0,1\}$, otherwise σ_p must be the inverse of $(\overline{F}^i)'_{kp}$. This restriction *(for all k)* will be used to construct the set of candidates of codes for the state yet to be encoded.

Secondly, consider for $i = m+1$ the k^{th} row of $F \wedge \overline{F}^i$:

$$(F \wedge \overline{F}^i)_{k\cdot} = ((F')_{k\cdot} \wedge \overline{(a \cdot \sigma)}_{k\cdot}) \ \vee \ ((a \cdot \sigma) \wedge \overline{(a \cdot \sigma)})_{k\cdot} =$$

$$= (F')_{k\cdot} \wedge \overline{(a_k \cdot \sigma)}$$

If $a_k = 1$ then a row of ϕ entries only occurs, otherwise at least one ϕ entry should occur, when σ is substituted.

In order to construct the set of candidates, the following variables have to be stored *statically*:

F'	the previous face matrix
$(\bar{F}^i)'$	the previous diminished inverted face matrices
$(\Phi^i)'$	the previous Φ-matrices

The algorithm then becomes:

```
$\sigma$ = (* * ... *) ;   /* All don't care values. */

for ($i$ = 1; $i \le m$; $i$++)
{
   for ($k$ = 1; $k \le n$; $k$++)
   {
      if ($a_k$ = 1)
      {
         determine $p$ such that $(\Phi^i)'_{kp} = \phi$ ;

         if ( $(\bar{F}^i)'_{kp} = \phi$ or $(\bar{F}^i)'_{kp} = *$ )

         /* $\sigma_p$ must be a Boolean ! */
         {
            candidateset = EMPTY ;
            stop ;
         }
         else
         /* $\sigma_p$ must become equal to $\neg(\bar{F}^i)'_{kp}$ */

         if ($\sigma_p$ = *)

            $\sigma_p = \neg(\bar{F}^i)'_{kp}$ ;

         else if ($\sigma_p = (\bar{F}^i)'_{kp}$)

         /* Contradiction ! */
         {
            candidateset = EMPTY ;
            stop ;
         }

      }

   }

}

$i = m + 1$;

for ($k$ = 1; $k \le n$; $k$++)

   if ($a_k$ = 0)

      check if at least one $\phi$ occurs when $\sigma$ is substituted ;


generate candidateset according to *-entries in $\sigma$ ;
```

The working of the algorithm is apparent: All necessary restrictions upon σ are stored by changing the corresponding *-entries into a 1 or a 0. At the end, the *-entries indicate the different possible codes σ can have. The candidateset can then be constructed by expanding the *-entries in σ into 1 and 0 respectively.

The other improvements mentioned in chapter 6 can also be applied to this algorithm. For instance, not all candidates have to be generated, but only a fraction should give good results; checking the face matrix first upon utilization of Boolean space makes it possible to have the algorithm generate only one candidate (with the lowest utilization); etcetera.

This supplement can be a good start for a practical traineeship to improve the program eskiss. A practical worker should firstly get acquainted with the theoretical basis of the algorithm KISS and the construction of the program eskiss. The thesis report will be an adequate piece of literature for this purpose. Then, the idea stated above to improve find_candidates must be investigated and implemented if found satisfactory.

Eindhoven University of Technology Research Reports
Department of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE

(147) Rozendaal, L.T. en M.P.J. Stevens, P.M.C.M. van den Eijnden
DE REALISATIE VAN EEN MULTIFUNCTIONELE I/O-CONTROLLER MET BEHULP VAN EEN GATE-ARRAY.
EUT Report 85-E-147. 1985. ISBN 90-6144-147-1

(148) Eijnden, P.M.C.M. van den
A COURSE ON FIELD PROGRAMMABLE LOGIC.
EUT Report 85-E-148. 1985. ISBN 90-6144-148-X

(149) Beeckman, P.A.
MILLIMETER-WAVE ANTENNA MEASUREMENTS WITH THE HP8510 NETWORK ANALYZER.
EUT Report 85-E-149. 1985. ISBN 90-6144-149-8

(150) Meer, A.C.P. van
EXAMENRESULTATEN IN CONTEXT MBA.
EUT Report 85-E-150. 1985. ISBN 90-6144-150-1

(151) Ramakrishnan, S. and W.M.C. van den Heuvel
SHORT-CIRCUIT CURRENT INTERRUPTION IN A LOW-VOLTAGE FUSE WITH ABLATING WALLS.
EUT Report 85-E-151. 1985. ISBN 90-6144-151-X

(152) Stefanov, B. and L. Zarkova, A. Veefkind
DEVIATION FROM LOCAL THERMODYNAMIC EQUILIBRIUM IN A CESIUM-SEEDED ARGON PLASMA.
EUT Report 85-E-152. 1985. ISBN 90-6144-152-8

(153) Hof, P.M.J. Van den and P.H.M. Janssen
SOME ASYMPTOTIC PROPERTIES OF MULTIVARIABLE MODELS IDENTIFIED BY EQUATION ERROR TECHNIQUES.
EUT Report 85-E-153. 1985. ISBN 90-6144-153-6

(154) Geerlings, J.H.T.
LIMIT CYCLES IN DIGITAL FILTERS: A bibliography 1975-1984.
EUT Report 85-E-154. 1985. ISBN 90-6144-154-4

(155) Groot, J.F.G. de
THE INFLUENCE OF A HIGH-INDEX MICRO-LENS IN A LASER-TAPER COUPLING.
EUT Report 85-E-155. 1985. ISBN 90-6144-155-2

(156) Amelsfort, A.M.J. van and Th. Scharten
A THEORETICAL STUDY OF THE ELECTROMAGNETIC FIELD IN A LIMB, EXCITED BY ARTIFICIAL SOURCES.
EUT Report 86-E-156. 1986. ISBN 90-6144-156-0

(157) Lodder, A. and M.T. van Stiphout, J.T.J. van Eijndhoven
ESCHER: Eindhoven SCHematic EditoR reference manual.
EUT Report 86-E-157. 1986. ISBN 90-6144-157-9

(158) Arnbak, J.C.
DEVELOPMENT OF TRANSMISSION FACILITIES FOR ELECTRONIC MEDIA IN THE NETHERLANDS.
EUT Report 86-E-158. 1986. ISBN 90-6144-158-7

(159) Wang Jingshan
HARMONIC AND RECTANGULAR PULSE REPRODUCTION THROUGH CURRENT TRANSFORMERS.
EUT Report 86-E-159. 1986. ISBN 90-6144-159-5

(160) Wolzak, G.G. and A.M.F.J. van de Laar, E.F. Steennis
PARTIAL DISCHARGES AND THE ELECTRICAL AGING OF XLPE CABLE INSULATION.
EUT Report 86-E-160. 1986. ISBN 90-6144-160-9

(161) Veenstra, P.K.
RANDOM ACCESS MEMORY TESTING: Theory and practice. The gains of fault modelling.
EUT Report 86-E-161. 1986. ISBN 90-6144-161-7

(162) Meer, A.C.P. van
TMS32010 EVALUATION MODULE CONTROLLER.
EUT Report 86-E-162. 1986. ISBN 90-6144-162-5

(163) Stok, L. and R. van den Born, G.L.J.M. Janssen
HIGHER LEVELS OF A SILICON COMPILER.
EUT Report 86-E-163. 1986. ISBN 90-6144-163-3

(164) Engelshoven, R.J. van and J.F.M. Theeuwen
GENERATING LAYOUTS FOR RANDOM LOGIC: Cell generation schemes.
EUT Report 86-E-164. 1986. ISBN 90-6144-164-1

(165) Lippens, P.E.R. and A.G.J. Slenter
GADL: A Gate Array Description Language.
EUT Report 87-E-165. 1987. ISBN 90-6144-165-X

(166) Dielen, M. and J.F.M. Theeuwen
AN OPTIMAL CMOS STRUCTURE FOR THE DESIGN OF A CELL LIBRARY.
EUT Report 87-E-166. 1987. ISBN 90-6144-166-8

(167) Oerlemans, C.A.M. and J.F.M. Theeuwen
ESKISS: A program for optimal state assignment.
EUT Report 87-E-167. 1987. ISBN 90-6144-167-6