A method for general simultaneous full decomposition of sequential machines: algorithms and implementation

Citation for published version (APA):

Document status and date:
Published: 01/01/1992

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

Please check the document version of this publication:

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

Link to publication

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

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

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

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 30. Mar. 2021
A Method for General Simultaneous Full Decomposition of Sequential Machines: Algorithms and Implementation

by

L. Józwiak
A.P.H. van Dijk

EUT Report 92-E-267
December 1992
A Method for General Simultaneous Full Decomposition of Sequential Machines: Algorithms and Implementation

by

L. Józwiaj and A.P.H. van Dijk
This report also stands for a report on the graduation work of A.P.H. van Dijk, performed by order of Prof. ir. M.P.J. Stevens and under supervision of Dr. ir. L. Jóźwiak.
A Method for the General Simultaneous Full Decomposition of Sequential Machines: Algorithms and Implementation

by Lech Józwia and Albert P.H. van Dijk.

Section of Digital Information Systems, Faculty of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

Abstract
Sequential machines that specify the control units and serial processing units of today's digital systems are quite complex and therefore designing, optimising, implementing and testing them is a complicated matter. This is the reason for the great interest in methods and CAD-tools for decomposing the complex machines into smaller and less complex ones.

One of the decomposition sorts feasible for sequential machines is a general simultaneous full decomposition. In this decomposition the sequential machine is decomposed into a number of simultaneously working partial machines which are able to interact, each with the other ones, and to realise together the behaviour of the original machine. There are no constraints on the direction of the interconnections between the partial machines and the total machine's behaviour is decomposed, i.e. not only the state set, but the input and output sets as well.

The aim of the reported research was to develop, implement and test an efficient heuristic method for the general simultaneous full decomposition of Moore machines with encoded inputs and outputs and symbolic states. It is required that the method will be able to discover the natural decompositional structure of the sequential machine.

As a result a method has been developed, implemented and tested which constructs the limited set of near-optimal decompositions using a beam search bottom-up hierarchical clustering algorithm, with decision making based on correlations between the information flows in the sequential machine.

In order to get an impression about the performance of the method, a number of experiments has been performed. In these experiments some machines have been used with a known optimal decompositional structure and other machines have been taken from the international benchmark set [18]. The preliminary results are promising, because in all the checked cases, the method was able to discover the natural decompositional structure by a very limited search; however, checking the method on larger samples of machines is still necessary in order to obtain more complete and more reliable information on the performance of the method.

Index terms: Automata theory, decomposition, logic design, sequential machines.

Acknowledgements: The authors are indebted to prof.ir. M.P.J. Stevens for making it possible to perform this work.
## Contents

1. Introduction 1

2. Partitions and partition pairs 5
   2.1. Symbolic partitions 5
   2.2. Bit partitions 7
   2.3. \( M() \) and \( m() \) operators 8

3. Decomposition 9
   3.1. The concept of decomposition 9
   3.2. The process of decomposing: hierarchical clustering 11

4. Algorithms 13
   4.1. Clustering 13
      4.1.1. Given a set of clusters 15
      4.1.2. Compute the correlation factors between each pair of clusters 15
      4.1.3. Determine per set of clusters a number of candidate clusters to be merged 19
      4.1.4. Are there any cluster sets which reached the final state? 19
      4.1.5. Construct candidate cluster sets by merging candidate cluster pairs 19
      4.1.6. Remove the superfluous and less promising candidate cluster sets 20
      4.1.7. Select the best cluster sets 20
   4.2. Analysis 21
      4.2.1. Compute \( S_{Rj} \) 23
      4.2.2. \( S_{Rj} = S_{Rj} \) 25
      4.2.3. Compute \( q_{x} \) for all pairs \( x \) and \( x_{i} \) in \( S_{Rj} \) 26
      4.2.4. Is \( S_{Rj} \) a good reduced input support set? 26
      4.2.5. Select the input bit with the lowest \( q_{x} \) and delete it from \( S_{Rj} \) 27
      4.2.6. Look at more possibilities for \( S_{Rj} \)? 27
      4.2.7. Determine which input support sets are suitable for the clustering algorithm 27
   4.3. Building the decomposition 28
   4.4. Don't care conditions 29

5. Implementation 31
   5.1. Using DEV 31
      5.1.1. System requirements 31
5.1.2. Installation and compilation 31
5.1.3. File formats 33
5.1.4. Running DEV 34
5.1.5. Changing the constants 36
5.2. The source code 36
  5.2.1. Data structures 36
  5.2.2. Program flow 38
5.3. Sources of the source code 45
6. Decomposition example 47
  6.1. Analysis 47
  6.2. Clustering 48
  6.3. Building the decomposition 50
7. Preliminary results 51
8. Conclusions 53

Literature 55

Appendices 57

Appendix 1: Configuration file 59
Appendix 2: Example run 63
Appendix 3: Include files 73
Appendix 4: Pseudo pascal description of the most important routines 77
1. Introduction

Methods and CAD-tools for decomposing complex digital systems and circuits have attracted great interest recently and there are several reasons for this:
- the extent and complexity of processes that must be implemented in the form of digital systems and circuits continues to grow rapidly,
- today's methods of design, optimisation, implementation and testing are not suitable for very complex circuits.

However, the strongest stimulus for developing decomposition methods and tools came recently from the field of programmable devices and, especially, from the newest generation of multi-block devices. Programmable devices impose hard constraints on the size of circuits which they implement. If the circuit to implement exceeds these constraints, decompositional implementation is not a choice but a real necessity.

The rapidly growing interest in programmable devices has been observed during the last few years and this tendency is predicted to continue. This is the result of some characteristic features of these devices and the growing number of applications which require these features. Programmable devices are characterised by the low cost, fast customisation (programmability) by their users, without the necessity for foundry involvement and by reprogrammability.

Compared to the semi custom or full custom circuits, they are better suited for system prototyping and small or medium volume production of application or field specific systems. Compared to the general purpose (micro-)computers they are better suited for the application in all the advanced systems where (re-)programmability and speed are required together.

However, the possibilities created by the programmable devices cannot be fully exploited at the moment because of the lack of the appropriate synthesis methods and tools. In this way, decomposition methods and tools have become very important in the digital system practice.

In order to solve or reduce the problems of complex systems and circuits, a structural decomposition approach can be used. It refers to transforming a system into a structure of two or more cooperating partial systems in such a way that the original system's behaviour is retained.

The theoretical work in this field was started by Hartmanis in the early 1960's. However, Hartmanis [3, 4] and others only considered the decomposition of the sequential machine's internal states and therefore it was not a complete solution. The most important design parameters of a circuit for implementing a sequential machine (complexity, speed, testability etc.) are functions of the entire implementation structure and depend both on the machine's internal states and its inputs and outputs. Furthermore, the possibility to implement a machine with limited building blocks depends on the number of internal states as well as the number of inputs and outputs and sometimes also on the other structural parameters (e.g. the number of product terms). So, from the practical viewpoint, decomposition of the entire functionality into an appropriate structure is
necessary, i.e. full decomposition.

The first concepts of full decomposition were presented in [1] and [2] by Y. Hou. The research work on full decompositions was continued within the section of Digital Information Systems, in the Department of Electrical Engineering at the Eindhoven University of Technology, resulting in a classification of possible full decompositions and theorems describing how to construct decompositions from each class [5, 6, 7]. Also the full decomposition concept was expanded to include the case of decomposing input and output bits, instead of symbolic inputs and outputs and the appropriate theorems for transforming the theory of symbol full decomposition into the theory of bit full decomposition were introduced [8]. The expanded decomposition theory constitutes a sound base for constructing practical decomposition algorithms for sequential and combinational circuits. In [5] the concept of a general full decomposition is introduced. General decomposition includes as its special cases various other decomposition types for sequential and combinational circuits. Among others, the parallel and serial decomposition are its special cases. Parallel, serial and general decompositions can be distinguished depending on the interconnections between the partial machines. In parallel decompositions, the machines don't interact. In serial decompositions, the state or output information flows in one direction, i.e. if the second partial machine receives information from a first one, no information flows from the second partial machine to the first one. In general decompositions, the state or output information can flow in both directions [5].

From the viewpoint of the interactions between the partial machines, two sorts of decompositions are feasible: simultaneous decompositions and sequential decompositions. Simultaneous decomposition decomposes a sequential machine into a number of interacting partial machines. All partial machines are simultaneously active. Sequential decomposition decomposes the machines into a number of interacting partial machines from which only one partial machine is active at any time.

For parallel and serial decompositions, we were able to develop and implement quite efficient decomposition algorithms, using only the algebraic properties which follow from the appropriate construction theorems [5, 6, 7, 8] and the partition pair theory of Hartmanis [4]. The research performed when using the software implementing these algorithms to decompose a benchmark set of 43 sequential machines (among others the machines from the international benchmark set [18]) has shown that useful parallel decompositions existed for 30% of the machines and useful serial decompositions for 50%. For 21% of the machines, neither serial, nor parallel decompositions existed for decomposing the internal states [9]. So in many practical cases, general full decompositions must be used.

For large sequential machines, the number of possible general decompositions is so large that the exhaustive search, limited only by the algebraic properties, is impossible. It becomes necessary to combine the appropriate general decomposition theorems [5, 6, 7, 8] and the partition pair theory with some heuristics in order to limit the search space to a manageable size. At the same time, the best solutions must be kept in this limited search space.
The aim of the research work, being the subject of this report, was to develop, implement and test an efficient heuristic method for the general simultaneous full decomposition of Moore machines.

Since in most practical cases the machines to be decomposed have assigned inputs and outputs and symbolic states, the research described in this report was limited to these machines.

An efficient method for constructing general simultaneous full decompositions of sequential machines with assigned states as well as a method for constructing general sequential full decompositions of sequential machines with symbolic states have been developed and implemented by us previously and are reported in [9] and [10].

In the work reported here, we limited our considerations to the Moore machines with multiple output bits; however the method developed can easily be extended to cover the Mealy machines with multiple output bits.

The method presented here is not suitable for decomposing the machines without or with only a small number of output bits. This limitation follows from the fact that the decomposition process performed by our method is based on the appropriate output partitioning. A similar method will be developed for decomposing the machines without or with only a few output bits. In this case, decomposition will have to be based on the input or state partitioning.

Nevertheless, the class of decompositions covered by the method is very important from the practical viewpoint, it includes, for example, large controllers which typically have very many output bits.

Summarising, our objective was to develop a decompositional method which should be able to discover the natural decompositional structure of a certain sequential machine, based on the analysis of some structural factors contained in the machine's definition. This definition can be given in the form of a state diagram or in the equivalent tabular form (KISS format). Such a method can be used for solving the decomposition problems without hard constraints and for the decompositional state assignment [10].

This report describes the developed method as well as its implementation and testing. In chapter 2, the necessary theoretical knowledge, which can also be found in [4, 5, 6, 7, 8], is summarised. Chapter 3 contains basic information about decompositions and how to construct them. In chapter 4, the exact algorithm is described and explained. Chapter 5 contains information about the implementation of the algorithm in the form of a computer program. An example machine is decomposed using the method in chapter 6 and the test results are summarised in chapter 7.
2. Partitions and partition pairs

2.1. Symbolic partitions

Theoretical tools used to decompose sequential machines are partition algebra and partition pair algebra. Below, only this information about partition and partition pairs is recalled, which is necessary to understand the method. All the definitions presented are from [4], [6] and [8].

A partition of a set $S$ is a set of subsets of $S$, referred to as blocks $B_k$, with

$$\forall k \neq 1 \quad B_k \cap B_1 = \emptyset \quad \land \quad \bigcup_k B_k = S. \quad (1)$$

A partition can be seen as a tool to provide some information about the elements of $S$. Knowing with which block from a partition over a set $S$ you are dealing with, doesn't mean that it is known which element of $S$ is dealt with. The uncertainty about the exact element increases with the magnitude of the block. So a partition with small blocks contains a lot of information and one with large blocks contains less.

Partitions can be multiplied and added:

$$s = t \{ \pi \cdot \tau \} \iff s \equiv t (\pi) \land s \equiv t (\tau)$$

$$s \equiv t (\pi + \tau) \text{ if there exists a sequence } s = s_1 \ldots s_k = t \text{ such that }$$

$$s_i \equiv s_{i+1} (\pi) \lor s_i \equiv s_{i+1} (\tau) \text{ for } 1 \leq i < k. \quad (2)$$

In (2), $\pi$ and $\tau$ are partitions on a set $S$; $s$, $t$ and $s_i$ are elements from $S$ and $s \equiv t (\pi)$ means that $s$ and $t$ are in the same block of partition $\pi$.

Multiplication intersects the blocks of the two partitions, and thus combines the information contained in the two partitions. Addition unites blocks which contain the same elements and is a representation for the information that is contained in one block that is also in the other block. Thus it represents the redundant information.

Partitions can be ordered using the rule

$$\pi > \tau \iff (s \equiv t (\tau) \Rightarrow s \equiv t (\pi)) \quad (3)$$
Thus \( \pi \supseteq \tau \) means that all blocks of \( \tau \) are contained in blocks of \( \pi \), or, to put it in another way, \( \tau \) contains all the information of \( \pi \), plus some extra information.

A useful measure for the amount of information contained in partition \( \pi \) is its modulus \(|\pi|\). \(|\pi|\) is the number of blocks of \( \pi \).

If a partition can describe information, it must also be possible to describe information flows. This can be done with partition pairs. To define this we first need a description of a sequential machine.

A sequential Moore machine \( M \) is a 5-tuple defined as follows

\[
M = (S, I, O, \lambda, \delta)
\]

with

- \( S \) : Set of symbolic states,
- \( I \) : Set of symbolic inputs,
- \( O \) : Set of symbolic outputs,
- \( \lambda : S \rightarrow O \) : The output function,
- \( \delta : I \times S \rightarrow S \) : The next state function.

Let \( \pi_s \) and \( \tau_s \) be partitions on \( S \), \( \pi_i \) a partition on \( I \) and \( \pi_o \) a partition on \( O \). Then \((\pi_s, \tau_s)\) is an \( S-S \) partition pair of machine \( M \) if and only if

\[
S \equiv \tau (\pi_s) \Rightarrow \forall \text{ inputs } i \quad \delta(i, s) \equiv \delta(i, \tau)(\tau_s).
\]

The interpretation of this partition pair is that the present state partition \( \pi_s \) combined with all the input information contains enough information to compute the next state partition \( \tau_s \). In a similar way \( I-S \) and \( S-O \) pairs can be defined and interpreted.

\((\pi_i, \pi_o)\) is an \( I-S \) partition pair if and only if

\[
i_1 \equiv i_2 (\pi_i) \Rightarrow \forall \text{ states } s \quad \delta(i_1, s) \equiv \delta(i_2, s)(\pi_i).
\]

And \((\pi_s, \pi_o)\) is an \( S-O \) partition pair if and only if

\[
s \equiv \tau (\pi_s) \Rightarrow \lambda(s) \equiv \lambda(\tau)(\pi_o).
\]
2.2. Bit partitions

All partitions above are partitions of symbolic sets. The machines we are working with in most cases have assigned input and output bits and symbolic states. In order to work with these input and output bits, bit partitions and bit partition pairs have been developed. A bit partition has one dcbr (don't care block) that contains all bits which are don't cares and a number of blocks each containing one single important bit (which is not don't care).

The definitions using bit partitions can be interpreted the same as the ones using symbolic partitions and are given below.

The modulus $|\pi_{B}|$ of a bit partition $\pi_{B}$ is the number of important bits in the partition.

Multiplication:

$$dcbr(\pi_{B1} \cdot \pi_{B2}) = dcbr(\pi_{B1}) \cap dcbr(\pi_{B2})$$

Addition:

$$dcbr(\pi_{B1} + \pi_{B2}) = dcbr(\pi_{B1}) \cup dcbr(\pi_{B2})$$

$S$ : Set of symbolic states,
$IB$ : Set of input bits,
$OB$ : Set of output bits,

$(\pi_{IB}, \pi_{OB})$ is an IB-S partition pair if and only if

$$\forall s \forall x \in dcbr(\pi_{B}) \delta(x_{1} \ldots x_{k-1}, 0, x_{k+1} \ldots x_{m}, s) \equiv \delta(x_{1} \ldots x_{k-1}, 1, x_{k+1} \ldots x_{m}, s) (\pi_{S})$$

7
with \( IB = \{x_1, x_m\} \) the set of input bits, \( x_k \) is 0 or 1 and \( s \) a symbolic state.

\((s,\pi_{OB})\) is an S-OB partition pair if and only if

\[
s \equiv t (\pi_s) \Rightarrow \forall y_j \in dcb(\pi_s) \lambda_j (s) = \lambda_j (t) \tag{10}
\]

with \( \lambda_j \) the output function of output \( y_j \), \( \lambda(s) = \{\lambda_j(s)\} \).

### 2.3. \(M()\) and \(m()\) operators

Finally the concept of \(M\) and \(m\) operators is used. These operators can be used with any kind of partition pairs. They will be defined using S-S partition pairs.

\( \pi = M_{s,s}(\tau) \) if and only if \( \pi \) is the largest partition such that \((\pi,\tau)\) is an S-S partition pair. Thus an \( M(\tau) \) gives a partition with which \( \tau \) can be computed and which contains minimal information to do so.

\( \tau = m_{s,s}(\pi) \) if and only if \( \tau \) is the smallest partition such that \((\pi,\tau)\) is an S-S partition pair. Thus \( m(\pi) \) computes the partition that contains the maximal amount of information that can be computed from \( \pi \).
3. Decomposition

3.1. The concept of decomposition

Decomposing is a well known problem solving principle. If there is a problem too difficult to solve at once, you can decompose it into smaller problems, which are more easy to solve. In order to solve the original problem, the solutions to the smaller problems must each provide a part of the solution and the partial solutions must be combined in the appropriate way. Sometimes one partial solution must be used to solve other partial problems. This provides for interaction between the partial solutions.

When designing sequential machines, we can use the same principle. If a machine is too complex to design, test, optimise or implement, it can be decomposed into smaller partial machines. These partial machines can each provide a part of the behaviour of the original machine and they can interact by exchanging state- or output information.

The decompositions we are looking for are simultaneous general full decompositions of type $S$. In these decompositions the partial machines communicate by exchanging some state information. This is shown in Figure 1 for a decomposition into two partial machines. Decomposing it into more than two partial machines, each partial machine can use some of the state information of all the other partial machines. The structure of the partial machines is given in Figure 2.

It should be clear that a machine cannot be decomposed randomly.

The partial machines have to fulfil certain criteria to make the decomposed machine behave the same as the original machine with regard to output behaviour.

The criteria the partial machines should comply with are (see [6, 8]) the following:
Figure 2: Structure of a partial machine.

For each $j$

\begin{align*}
(p_{IBj}, p_{Sj}) & \text{ is an IB-S partition pair,} \\
(p_{Sj}, \tau_{Sj}, p_{Sj}) & \text{ is an S-S partition pair and} \\
(p_{Sj}, \tau_{Sj}, \pi_{obj}) & \text{ is an S-OB partition pair.} \\
\end{align*}

(11)

with $\pi_{sp}, p_{IBj}, \pi_{obj}$ and $\tau_{Sj}$ respectively the state partition, the input bit partition, the output bit partition and the returned state partition from other partial machines.

Our objective is to construct decompositions which satisfy (11), in which the complexity of each partial machine is in each case lower than that of the original machine and the number of interconnections between the partial machines is minimal. We will thus try discover the natural decompositional structures in the machine.
3.2. The process of decomposing: hierarchical clustering

To come to a useful decomposition, an appropriate partition $\mu_{OB}$ on the set of output bits $OB$ has to be formed. Output bits which are in the same block in this partition, will be realised in the same partial machine.

We form this output partition to emphasise the input-output dependence. Other methods, which can rely on forming state partitions, emphasise the state-dependence within each partial machine. Since the number of state bits used in each partial machine is $2^{\log |x_{in}|}$ and the number of input bits $|x_{in}|$, input dependence will be more important for larger machines than state dependence. It is better to add a few state-blocks if you can save one or two input bits for a partial machine in doing so.

To construct the output partition $\mu_{OB}$ we will use the concept of hierarchical clustering [19]. In this concept a cluster is a block of the partition $\mu_{OB}$. A near optimal partition can be constructed in two ways (see also Figure 3):

1. Starting with all output bits in one cluster ($\mu_{OB} = \mu(I)$), a cluster can be selected and split in two clusters. This splitting makes the clusters become smaller and can be repeated until an optimal partition is obtained.
2. Starting with each output in a single cluster ($\mu_{OB} = \mu(0)$), a pair of clusters can be selected to be merged. This merging makes the clusters larger and can be repeated until an optimal partition is obtained.

The first method is referred to as top-down hierarchical clustering, the second one as bottom-up hierarchical clustering. Top-down clustering starts with splitting the machine into large parts and

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{hierarchical_clustering.png}
\caption{Hierarchical clustering.}
\end{figure}
therefore is more liable to discover the global structures of the machine. Bottom-up clustering will look at the more local structure of the machine first.

Both bottom-up and top-down clustering will lead, if implemented ideally, to the optimal solution. We will use bottom-up hierarchical clustering. Top-down clustering is more complex. The number of possibilities to split a cluster containing n elements is $2^{n-1}$ and the number of possibilities to merge two from n clusters is $\frac{1}{2} n (n-1)$. So for larger machines it is not possible to look at all possibilities to split a cluster, while it remains possible to look at all the merges of two clusters. When heuristics are used to build clusters, top-down clustering can be used, but the heuristics for bottom-up clustering are less complex.

Using bottom-up hierarchical clustering the problem arises which two clusters should be merged. A quantitative measure should be found which estimates the probability that merging two clusters leads to an optimal partition. A measure that can be used is the correlation between the information that two clusters need to compute the next state and output. Clusters with high correlation should be merged. Both using the same information, the amount of information that is needed as an input and the amount of logic to process the information will be minimal if these clusters are merged.

Merging clusters cannot go on forever, at a certain moment all clusters will have merged to form one big cluster. Therefore it has to be detected when an optimal cluster set, or partition is reached and no more clusters should be merged.

This decision depends on the optimisation criterium that is used. Here we look for a decomposition into largely independent partial machines, which are each less complex than the original machine. The merging should therefore stop when there is little information left to merge any more clusters. This means that the correlation factors between the clusters should all be low.

This can of course be combined with constraints such as a minimal or maximal number of clusters, a minimum or maximum size of clusters etc.

After finding an optimal cluster set the decomposition is nearly complete. The output partition is computed. During clustering, the input partition $\pi_{b_j}$ is constructed for each partial machine and so are the state partitions $\pi_{s_j}$ and the returned state partitions $s_{s_j}$. All that needs to be done is to determine from which partial machine(s) the state partitions $s_{s_j}$ are coupled back. Once these connections are made the original machine is decomposed into a number of communicating partial machines.

The decomposed machine still possesses symbolic states and cannot be implemented yet. State assignment of each partial machine maps these symbolic states on state bits and then the logic functions for each input- and state bit can be implemented.
4. Algorithms

Since we want to use the clustering algorithm to compute the output partition, the entire algorithm will be divided in three steps:

1. Analysis.
2. Clustering.
3. Building the decomposition.

In the first step the information that is needed in the clustering algorithm is computed. In the third step the cluster(s) formed by the clustering algorithm will be formed into actual decompositions and the final state assignment will be done.

The clustering algorithm is central in the entire process. It will be described first. In the second section the algorithms for the analysis will be dealt with and in the third the building of the decomposition will be described.

4.1. Clustering

The clustering process consists of choosing two clusters to be merged, merging them and then determining whether the obtained cluster set is optimal. If it is not optimal, the process will be repeated.

If we want to use the algorithm like this, the probability that merging the two chosen clusters leads to finding the optimal solution must be equal to one. This probability will in most cases not be equal to one. So to find the optimal partition with certainty all possible clusterings should be checked. Because this can not be done for computational reasons, we will use a beam search algorithm.

In a beam search not all possible merges are performed, as it is shown in Figure 4. In this figure a circle is a cluster set, a line is a possible merge. Instead of performing all possible merges, only a limited number of merges is performed per cluster set. In the entire tree of possibilities only a part, a beam, is evaluated. This beam should be that part of the tree that has the highest probability to contain an optimal solution and is still computable.

A flow diagram of a beam-search algorithm computing cluster sets is shown in Figure 5. Each block of the diagram will be explained below.
Input a cluster set.

Compute the correlation factors between each pair of clusters.

Determine per cluster set a number of candidate cluster pairs to be merged.

Cluster sets with the final state?

Save those cluster sets which reached the final state.

Any cluster sets left?

Select best cluster set(s).

Construct candidate cluster sets by merging candidate cluster pairs.

Remove superfluous and less promising candidate cluster sets.

Call candidate cluster sets normal cluster sets.

Figure 5: The beam search algorithm.
4.1.1. Given a set of clusters

Input of the algorithm is a cluster set holding enough information to compute the correlation factors. Cluster \( j \) will therefore have to contain an input partition \( \pi_{s Bj} \), a state partition \( \pi_{s Bj}^3 \) and an output partition \( \pi_{o Bj} \). Furthermore it will contain a list of demands for the returned state partition \( \tau_{3j} \). These demands are stored as a set \( T_j \) of state pairs \((s,t)\) which are not allowed to be in the same block in \( \tau_{3j} \). The cluster set we start with will be a cluster set with each cluster containing exactly one output.

4.1.2. Compute the correlation factors between each pair of clusters

We will take one cluster set at a time. For each cluster set we will look at all possible combinations of two clusters \( i \) and \( j \), \( i > j \). For each combination of clusters \( i \) and \( j \) a correlation factor will be computed. This correlation factor gives information about the correlation between two clusters in two or more dimensions. A correlation factor is a number between 0 and 1. Pairs of clusters with high correlation factors can be merged. First of all, the methods to compute the correlation between two bit partitions or between two symbolic partitions will be introduced. Then the different dimensions in which the correlation factors will be computed and the method to compute the correlation factors in these dimensions will be discussed. In the last section, a method to combine the different dimensions is given.

4.1.2.1. Correlation between bit partitions

Correlation between bit partitions will be used for input bit partitions. It can be computed when using a set description. We will use a set \( S_{ri} \) to denote the reduced input support set of the cluster \( i \). This set contains all important input bits of the bit partition, i.e. all input bits that are not in the don't care block of the input bit partition.

There are two alternatives to compute the correlation between the input support sets \( S_{ri} \) and \( S_{rj} \). The set structure is shown in Figure 6.

In the first alternative, the difference between the two cluster sets in relation with their total magnitude is minimised. Note that we have chosen to minimise the difference in relation to their total magnitude. Minimising the absolute difference can be used, but it will have high correlation for partitions with a large number of elements. This would, while clustering, lead to the creation of clusters.

Figure 6: Set structure.
containing a large amount of information, not merging clusters with less information.

The difference between the cluster sets are the regions B and C in Figure 6a, the total magnitude the regions A, B and C. A formula to compute this correlation factor \( c_A \) is:

\[
\frac{|S_{Ai} \cap S_{Aj}|}{|S_{Ai} \cup S_{Aj}|}.
\]

(12)

where region A is represented by the intersection of \( S_{Ai} \) and \( S_{Aj} \) and regions A, B and C by the union of the sets.

The second alternative is illustrated in Figure 6 b. The number of input bits of the smaller cluster, which are not used by the larger cluster, is minimised. The area to be minimised is region C in Figure 6 b. It will be minimised related with areas A and C. A formula to compute this correlation factor \( c_B \) is:

\[
\frac{|S_{Ai} \cap S_{Aj}|}{\min(|S_{Ai}|, |S_{Aj}|)}.
\]

(13)

From Figure 6, it can be seen that the first alternative gives high correlation between input support sets with about the same magnitude, which have a lot of bits in common. The second alternative can be used to look with which larger partition smaller partitions can be merged in the best way.

4.1.2.2. Correlation between symbolic partitions

To compute the correlation between the symbolic state partitions \( \pi_{si} \) and \( \pi_{sj} \) we will use the same concepts we used to compute the correlation between sets of input bits. Area A in Figure 6 is maximised. It represents the information that is in both partitions \( \pi_{si} \) or \( \pi_{sj} \), i.e. the redundant information in both partitions. With symbolic partitions this redundant information is given by the addition of the partitions.

Again there are two alternatives to compute the correlation between two state partitions.

In the first case the redundancy is maximised with respect to the total amount of information in \( \pi_{si} \) and \( \pi_{sj} \) the correlation factor \( c_{x1} \) is defined as

\[
\frac{|\pi_{si} + \pi_{sj}|}{\pi_{si} \cdot \pi_{sj}}.
\]

(14)

with the sum of the partitions representing the region A and the product representing the regions A, B and C.
Using the second alternative, the region B or C is minimised and the correlation factor $c_{x2}$ is given by:

$$
c_{x2} = \frac{|\pi_{x2}^s|}{\min (|\pi_{s1}^s|, |\pi_{s2}^s|)}.
$$

(15)

Both alternatives can be interpreted in a way similar to the two alternatives in computing the correlation between bit partitions; the first alternative will give high correlation for partitions which contain about the same amount of information and the second alternative will do so for partitions with little information, which is also contained in a partition with more information.

4.1.2.3. Correlation dimensions

There are different dimensions in which clusters can correlate. High correlation is required in the dimensions to be optimised. Since we want to minimise the complexity of the entire machine, this complexity must be described by one or more dimensions.

High complexity of the entire machine results from a lot of connections between the partial machines or from complex partial machines. So, we have two dimensions in which the machines can be minimised.

The first dimension is the amount of connections between the partial machines. This can be minimised by minimising the number of inputs of each partial machine. It can be seen from Figure 2 that the input of partial machine $M_j$ consists of two partitions: $\pi_{Bj}$ and $\tau_S$. So, the first dimension of the correlation between the clusters $i$ and $j$ is the correlation between $\pi_{Bi}$ combined with $\pi_{Bj}$ and $\tau_S$ combined with $\tau_S$. For computing the correlation factor we can use the two alternatives presented in 4.1.2.1 and in 4.1.2.2. The correlation factor $c_{1,ij}$ between cluster $i$ and $j$ in the first dimension is, using the first alternative:

$$
c_{1,ij} = \frac{|S_{Ri} \cap S_{Rj}| + 2\log |\tau_{si} + \tau_{sj}|}{|S_{Ri} U S_{Rj}| + 2\log |\tau_{si} \cdot \tau_{sj}|}
$$

or, using the second alternative,

$$
c_{1,ij} = \min ( |S_{Ri}| + 2\log |\tau_{si}|, |S_{Rj}| + 2\log |\tau_{sj}| )
$$

(16)

Note that since $\tau_S$ is a symbolic partition the minimal number of bits it needs in the implementation is $2\log |\tau_S|$. $\pi_B$ is a bit partition and therefore it needs $\log |\pi_B|$ bits in the implementation.

The second dimension in the total complexity is the complexity of the partial machines. This can
be expressed in the amount of logic, needed to implement the next state function $\delta$ and the output function $\lambda$ and in the amount of flip-flops needed to store the present state.

Using PLA building blocks to implement the functions, the complexity can be estimated with $(2i + o)$, where $i$ is the number of input bits of the function and $o$ is the number of output bits. From Figure 2 can be seen that the next state function $\delta$ has three partitions as an input: The bit partition $\pi_m$ and the symbolic partitions $\pi^s_s$ and $\tau_s$. $\delta$ has one symbolic partition as an output, $\pi^s_s$. In the case of a Moore machine, the output function $\lambda$ has two symbolic partitions as an input, $\pi^s_s$ and $\tau_s$. It has the output partition $\pi_{0b}$ as an output, but this partition doesn't influence the complexity of the entire machine because each output bit has to be implemented in exactly one partial machine.

Combining all this information we can come to a correlation factor $c_{2ij}$ for the correlation between clusters $i$ and $j$ in the second dimension. This correlation factor can also be computed using the two alternatives presented in 4.1.2.1 and in 4.1.2.2. For the alternative 1, the correlation factor $c_{2ij}$ is:

$$
c_{2,ij} = \frac{2 \cdot | S_{ri} \cap S_{rj} | + 4 \cdot 2^{\log \tau_{si}} + \tau_{sj} | + (5 + F) \cdot 2^{\log \tau_{ss}} + \tau_{ss} |}{2 \cdot | S_{ri} \cup S_{rj} | + 4 \cdot 2^{\log \tau_{si}} + \tau_{sj} | + (5 + F) \cdot 2^{\log \tau_{ss}} + \tau_{ss} |}
$$

and for the second alternative

$$
c_{2,ij} = \frac{2 \cdot | S_{ri} \cap S_{rj} | + 4 \cdot 2^{\log \tau_{si}} + \tau_{sj} | + (5 + F) \cdot 2^{\log \tau_{ss}} + \tau_{ss} |}{\min_{k \neq i,j} (2 \cdot | S_{rk} | + 4 \cdot 2^{\log \tau_{sk}} + \tau_{sk} | + (5 + F) \cdot 2^{\log \tau_{ss}} + \tau_{ss} |)}
$$

In this equation, $F$ is a scaling factor with which the influence of the number of flip-flops used to store the state bits on the complexity is determined. The higher $F$, the larger the influence of the flip-flops is.

In this way, we are able to compute the cluster's correlation in both dimensions.

4.1.2.4. Combining dimensions

To compare the correlation factors, the two correlation factors computed in the two dimensions in 4.1.2.3 will be combined to form one correlation factor. This can be done by simply adding them using some scaling factors. The factor $c_{ij}$ for the correlation between clusters $i$ and $j$ is

$$
c_{ij} = a_1 \cdot c_{1,ij} + a_2 \cdot c_{2,ij}
$$

with $a_1$ and $a_2$ scaling factors which can be chosen by the user to emphasise either optimisation of the number of connections or the complexity of the partial machines.

One remaining problem is the fact that two alternatives were suggested to compute the correlation
factors. Both alternatives can be used in the algorithm. From Figure 6, it can be seen that the first alternative is best suited to merge clusters of about the same size, which have a lot in common, while the second alternative merges small clusters to the best large cluster. In the algorithm, first the first alternative can be used to find the global structures of the machine and after this the second alternative can be used to limit the number of small clusters by merging them with larger clusters.

4.1.3. Determine per set of clusters a number of candidate clusters to be merged

This step determines which branches are taken in the beam search. Only the branches with the highest correlation factors are selected. To limit the beam, there are two restrictions. First the number of branches that can be chosen per cluster set is limited to a maximum value. Also selected branches need to have a correlation higher than a certain minimum value.

4.1.4. Are there any cluster sets which reached the final state?

Cluster sets have reached the final state when there is little information left to merge any more clusters. At this time, the natural structure of the machine is found; it is decomposed into largely independent partial machines.

There is little information left to merge any more clusters if all cluster pairs have almost the same correlation or if all correlations are very low. In the first case, a decomposition in functional blocks is reached, in the second case merging any more clusters only makes the partial machines to be realised more complex, gaining little or nothing in amount of logic.

These criteria are used together with such criteria as maximum or minimum magnitude of clusters and maximum or minimum number of clusters in one cluster set.

When cluster sets have reached a final state they are moved to the list of completed sets.

4.1.5. Construct candidate cluster sets by merging candidate cluster pairs

Merging clusters i and j produces a cluster ij with the following properties.

\[ \pi_{IBij} = \pi_{IBi} \cdot \pi_{IBj} , \]
\[ \pi_{OBJij} = \pi_{OBI} \cdot \pi_{OBJ} , \]
\[ \pi_{Sij}^s = m (\pi_{IBij}) \]

and demands for \( \tau_{sij} \) such that

\[ \pi_{Sij}^s \cdot \tau_{sij} \leq M (\pi_{Sij}) \cdot M (\pi_{OBI}) . \]
So the algorithm first takes the input and output bits of cluster i and j together. Then it computes the minimal state partition (with the maximum amount of information) that can be computed from the input support. Finally it constructs a list of demands for the returned state partition $\tau_{sj}$ such that the next state partition and the output partition can be computed.

### 4.1.6. Remove the superfluous and less promising candidate cluster sets

It can happen that there are two or more candidate cluster sets with the same clusters. In this case one or more cluster sets are superfluous and can be deleted. This prevents the same computations to be done twice.

Deleting the less promising candidate cluster pairs is a part of the beam search algorithm. It is a way to control the width of the beam. If the beam is too wide, i.e. it contains too many branches, it is very likely that many cluster sets will lead to some near optimal decompositions. A number of candidate cluster sets can be deleted without too high a risk of not finding a near optimal solution. This will limit the amount of search that needs to be done.

A quality estimator must be found to determine which candidate cluster set should be deleted. As a quality estimator we will use the sum of the correlation factors of the merges made. Thus the history of constructing the candidate cluster set is taken into account. If the candidate cluster set was computed by merging clusters that had a high correlation, it is very likely that the candidate cluster set will lead to a near optimal decomposition. If a number of merges, on the other hand, had relatively low correlations, the decomposition it will lead to can still be near optimal, but it will probably not be as good as for the other candidate cluster sets.

So, if the number of candidate cluster sets reaches a certain limit, the set of candidate cluster sets can be reduced by using this quality factor.

### 4.1.7. Select the best cluster sets

After all cluster sets have either been removed or moved to the completed list, the clustering algorithm has ended and a number of good cluster sets are stored in the completed list. The best clusters from this list should be chosen for building the decomposition.

To choose the best cluster set, a quality estimator is needed. Using the two dimensions of complexity defined in 4.1.2, a quality estimator $q_j$ for cluster j can be defined

\[
q_j = a_1 \cdot \frac{|S_{kj}| + \log |\tau_{sj}|}{2} + a_2 \cdot \frac{2 \cdot |S_{kj}| + 4 \cdot \log |\tau_{sj}| + (5+F) \cdot 2 \log |\pi_{sj}|}{1 + F}
\]

with $a_1$, $a_2$ and $F$ the constants used in 4.1.2.3. The denominators in the divisions give the same
weight to both dimensions. One of the dimension can be emphasised by using the factors \(a_1\) and \(a_2\). The quality of a cluster set can be defined as the average quality of its clusters.

### 4.2. Analysis

In the analysis phase the input data for the clustering algorithm should be computed. This data is in the form of a cluster set with each cluster containing one output. Cluster \(j\) must contain an input partition \(\pi_{i,j}\), a state partition \(\pi^s_{s,j}\) and an output partition \(\pi_{o,j}\). Furthermore it must contain a list of demands for the returned state partition \(\tau_{s,j}\). These demands are stored as a set \(\mathcal{T}_j\) of state pairs \((s,t)\) which are not allowed to be in the same block in \(\tau_{s,j}\). The partitions must comply with the criteria stated in (11). These equations will first be rewritten using \(M()\) and \(m()\) partitions. Doing this simplifies computing the demands for various partitions.

First of all we will look at a way to compute the IB-S partition pair from (11). If \(\pi_{a_1} + \pi_{a_2} = \pi_{a_3}\), \(m_{IB,S}(\pi_{a_3}) = m_{IB,S}(\pi_{B1}) + m_{IB,S}(\pi_{B2})\). So, the \(m_{IB,S}\) partitions can be computed by adding \(\pi^c_s(x_k)\) partitions, with

\[
\pi^c_s(x_k) = m(\omega_k) \quad \text{and} \quad dcb(\omega_k) = (x_k) .
\]

where the \(\text{dc}\) stands for don't care. \(\pi^c_s(x_k)\) is the state information that can be computed if the input bit \(k\) is don't care and the rest of the input bits is known. The partition \(m_{IB,S}(\pi_{IB})\) can now be computed using the following formula:

\[
m_{IB,S}(\pi_{IB}) = \sum_{x_k \in dcb(\pi_{IB})} \pi^c_s(x_k) .
\]

Now (11) can be rewritten when using \(M_{S,S}\), \(M_{S-OB}\) and \(\pi^c_s(x_k)\) partitions.

\[
\pi^s_{s,j} \geq \sum_{x_k \in s_{s,j}} \pi^c_s(x_k) \quad \text{(23)}
\]

\[
\pi^s_{s,j} \cdot \tau_{s,j} \leq M_{S-S}(\pi^s_{s,j}) \cdot M_{S-OB}(\pi_{o,j}) .
\]

where \(S_{s,j}\) is the reduced input support set. This set contains those bits of the input bit set IB that are not in the don't care block of \(\pi_{i,j}\), i.e. the important input bits.

From (23) the effect of reducing the input support can be seen. If less input bits are used then \(S_{s,j}\) gets smaller and not as much state information \(\pi^s_{s,j}\) can be computed; therefore \(\tau_{s,j}\) should contain
more information.
The objective of reducing the input support set is to minimise the number of input bits that each output needs, while keeping the number of blocks in the returned state partition \( \tau_{ij} \) limited. To find the optimal reduced input support set, all possible subsets of the input set \( IB \) should be considered. This can computationally not be done. To limit the computational difficulty the primary input support set \( S_I \) is computed first. Input bits which are definitely not needed in cluster \( j \) are not in \( S_I \). Then input bits can be deleted from this \( S_I \), to form a reduced input support set with a compromise between the number of input bits and the amount of returned state information in \( \tau_{ij} \).

Two ways to compute the reduced input support set \( S_R \) from this set \( S_I \) without looking at all possible subsets were found:

1. Starting with an empty input support set, some important input bits from \( S_I \) can be added to \( S_R \). This will be continued until an optimal \( S_R \) is computed.
2. Starting with \( S_R = S_I \) and the less important bits from \( S_R \) are deleted until an optimal solution is reached.

Which method will be chosen depends on which method gives the most accurate estimation of the consequences of changing \( S_R \). It can be illustrated that method 1 is less accurate than method 2.

Method 1:
Adding one bit \( x_k \) to an existing input support set \( S_R \) means that two partitions have to be multiplied. One partition, \( \pi_{IB1} \), containing \( x_k \) in the important block and the rest in the don’t care block. The other partition, \( \pi_{IB2} \), containing the input bits from \( S_R \) in the important blocks and the rest of the input bits in the don’t care block. The amount of state information that can be computed from these partitions separately is \( m_{IB} (\pi_{IB1}) \) and \( m_{IB} (\pi_{IB2}) \). Since

\[
m_{IB} (\pi_{IB1}) \cdot m_{IB} (\pi_{IB2}) \geq m_{IB} (\pi_{IB1} \cdot \pi_{IB2}) \quad \text{and so}
\]

\[
M_{IS} (m_{IB} (\pi_{IB1} \cdot \pi_{IB2})) \leq M_{IS} (m_{IB} (\pi_{IB1}) \cdot m_{IB} (\pi_{IB2})) = M_{IS} (m_{IB} (\pi_{IB1})) \cdot M_{IS} (m_{IB} (\pi_{IB2}))
\]

(24)

the state partition that can be computed by adding \( x_k \) to \( S_R \) can only be estimated, unless the partitions are actually multiplied and the \( m_{IB} \) partition is computed. The \( M_{IS} \) partition can be computed exactly, but the argument is not exact.

Method 2:
The consequences of deleting input bit \( x_k \) from the input support set \( S_R \) must be estimated. Again there are two input bit partitions, \( \pi_{IB1} \) and \( \pi_{IB2} \). \( \pi_{IB1} \) contains only \( x_k \) in the don’t care block. \( \pi_{IB2} \) has all input bits from \( S_R \) in the important blocks, the rest of the bits in the don’t care block.

22
Deleting bit \( x \) from the input support set \( S_{Rj} \) is equivalent to adding partitions \( \pi_{IB1} \) and \( \pi_{IB2} \). Since

\[
m_{IB-S}(\pi_{IB1}) + m_{IB-S}(\pi_{IB2}) = m_{IB-S}(\pi_{IB1} + \pi_{IB2}) \quad \text{and so}
\]

\[
M_{S-S}(m_{IB-S}(\pi_{IB1}) + m_{IB-S}(\pi_{IB2})) \geq M_{S-S}(m_{IB-S}(\pi_{IB1})) + M_{S-S}(m_{IB-S}(\pi_{IB2}))
\]

(25)

the state partition that can be computed by deleting \( x \) from \( S_{Rj} \) can be determined by adding the \( m_{IB-S} \) partitions. Forming the S-S partition pair however the \( M_{S,S} \) partition has to be estimated.

Because the estimation with method 1 is made in the earlier computation stage, the choices based on this estimation are expected to be less accurate. Therefore, method 2 is used.

A choice has to be made which input bit should be deleted from \( S_{Rj} \). This choice can be made by deleting input bit \( x \) from the \( S_{Rj} \) and looking at the consequences. If not too much information is lost, that is if \( \tau_{Sj} \) doesn’t increase too much, the input bit \( x \) could be deleted from \( S_{Rj} \).

The consequences of deleting the input bit \( x \) from \( S_{Rj} \) can only be estimated. This estimation can be made more accurate by looking at larger groups of input bits. Thus we will use groups of two bits \( x \) and \( x_i \) to determine which input bits should be deleted. If input bit \( x \) is not relevant for output \( j \) the consequences of deleting \( x \) and \( x_i \) from \( S_{Rj} \) will be low for all \( i \).

The consequences of deleting input bits \( x \) and \( x_i \) from the input support set can be expressed in a quality factor \( q_{Sj} \). If \( q_{Sj} \) is low, not much information is lost by deleting \( x \) and \( x_i \). The quality of an input bit \( x \) can then be computed by adding the quality factors \( q_{Sj} \) for all \( i \).

Method 2 is implemented in the block diagram in Figure 7. The implementation of the separate blocks will be discussed below.

4.2.1. Compute \( S_{Pj} \)

The primary set of inputs contains the inputs that are necessary to compute the output bit \( j \). With this set of inputs \( \tau_{Sj} = \tau_{S1} \) (contains no information) and only the IB-S and S-OB partition pairs from (11) are realised.

The input and output partitions are such that

\[
dcb(\pi_{obj}) = \{ y_k \mid y_k \neq y_j \}
\]

\[
dcb(\pi_{IBj}) = \{ x_k \mid x_k \in S_{Pj} \}. \tag{26}
\]

The demands for the primary input support set \( S_{Pj} \) should be such that two partition pairs exist
Figure 7: Output bit analyses.
\((\pi_{IBj}, \pi_{Sj})\) is an IB-S partition pair and
\((\pi_{Sj}, \pi_{OBJ})\) is an S-OB partition pair.

The demands for the partition pairs can be rewritten, using the \(\text{M}()\) and \(\text{m}()\) operators, as

\[
\begin{align*}
\pi_{Sj}^S &\geq \text{m}_{IB-S}(\pi_{IBj}) \\
\pi_{Sj}^S &\leq \text{M}_{S-OB}(\pi_{OBJ})
\end{align*}
\]

So, from (28) the demand for the input partition is

\[m_{IB-S}(\pi_{IBj}) \leq M_{S-OB}(\pi_{OBJ}).\]  

Using (26) and (23) the amount of state information that can be computed from the primary input support set is

\[m_{IB-S}(\pi_{IBj}) = \sum_{x_k \in S_{pj}} \pi_{Sj}^{dc}(x_k).\]  

And from (29) and (30) the primary input support set can be computed

\[S_{pj} = \{x_k | \pi_{Sj}^{dc}(x_k) \leq M_{S-OB}(\pi_{OBJ})\}.\]

with \(\leq\) meaning not smaller or equal (which is not the same as > with partitions).

This last equation can easily be implemented to compute \(S_{pj}\).

4.2.2. \(S_{Rj}=S_{pj}\)

Now the reduced input support set \(S_{Rj}\) is initialised. The state partition \(\pi_{SPj}\) and the returned state partition \(\tau_{SPj}\) must be initialised to comply with (23). They are initialised according to
\[ S_{Rj} = S_{Rj} \]
\[ \pi_{s0} = \sum_{x_k \in S_{Rj}} \pi_{dc}(x_k) \]  \hspace{1cm} (32)

Demands for \( \tau_{s0} \) such that
\[ \pi_{s0} \cdot \tau_{s0} \leq M_{s-s}(\pi_{s0}) \cdot M_{s-ob}(\pi_{obj}) \]

**4.2.3. Compute \( q_{ki} \) for all pairs \( x_k \) and \( x_i \) in \( S_{Rj} \)**

The quality factor \( q_{ki} \) reflect the consequences of deleting \( x_k \) and \( x_i \) from \( S_{Rj} \).

Deleting input bits results in the increase of \( \tau_{s0} \) (see (23)). The information that is lost by deleting the input bits has to be compensated by adding some information to \( \tau_{s0} \). The less \( \tau_{s0} \) increases, the less important the two input bits are for output \( y \).

The state partition \( \pi_{dc_{sk}} \) that can be computed when \( x_k \) and \( x_i \) are deleted from \( S_{Rj} \) is

\[ \pi_{dc_{sk1}} = \pi_{s0} + \pi_{dc}(x_k) + \pi_{dc}(x_i) \]  \hspace{1cm} (33)

and the returned state partition \( \tau_{sk1} \) that is needed to satisfy (23) can be computed using

\[ \pi_{dc_{sk1}} \cdot \tau_{sk1} \leq M_{s-s}(\pi_{s0}) \cdot M_{s-ob}(\pi_{obj}) \]  \hspace{1cm} (34)

Now the quality factor \( q_{ki} \) is defined as the increase in the number of blocks of \( \tau_{s0} \), thus reflecting the extra amount of information needed.

\[ q_{sk1} = | \tau_{sk1} | - | \tau_{s0} | \]  \hspace{1cm} (35)

The higher the quality factor is, the more important the input bits are for computing the value for a certain output bit.

After computing \( q_{ki} \), the quality factor \( q_{k} \) for the input bit \( x_k \) can be computed by summing the quality factors \( q_{ki} \) for all \( i < k \).

**4.2.4. Is \( S_{Rj} \) a good reduced input support set?**

The reduced input support set is good when there is little information left to delete a successive input bit. This does not mean that one has to stop the further search, because it is often possible to
find a better input support set. But it does mean that a more or less coherent group of input bits is obtained.

Little information is left to delete another input bit when a number of bits have almost the same quality factor. Additionally, this quality factor should be low and the group of bits must be large enough.

4.2.5. Select the input bit with the lowest $q_k$ and delete it from $S_{Rj}$

In this step the problem is not in selecting an input bit, but in deleting it. Suppose input bit $x_k$ is selected. Then it is first deleted from $S_{Rj}$ and after this $\pi^3_{sj}$ and $\tau_{sj}$ will be adjusted to comply with (23).

$$S_{Rj} = S_{Rj,old} \setminus x_k$$

$$\pi_{s0} = \pi_{s0,old} + \pi^i_{s0}(x_k)$$

and $\tau_{s0}$ such that $\pi_{s0} \cdot \tau_{s0} \leq M_{s-o}(\pi_{s0}) \cdot M_{s-o}(\pi_{s0})$

(36)

4.2.6. Look at more possibilities for $S_{Rj}$?

Other possibilities for the reduced input support set can be looked at as long as $S_{Rj}$ still contains input bits. Other criteria to stop the search can be used as well. The search can for example be stopped when too much returned state information must be added when any more bits are deleted from the reduced input support set.

Since the time it takes however to compute the deletion of the last few bits from the reduced input support set $S_{Rj}$ is relatively small other criteria don’t have to be used.

4.2.7. Determine which input support sets are suitable for the clustering algorithm

With a few minor adjustments to the clustering algorithm, it can work with cluster sets which contain one and the same output bit partition in more than one cluster. A partial machine which realises certain output bits can thus have more than one option for the input support set $S_{Rj}$ and for the state partitions $\pi^3_{sj}$ and $\tau_{sj}$. These options are stored in different clusters with the same output bit partition. When clusters are merged, the best option for a certain output bit partition is chosen and the rest of the options with that output bit partition are deleted.

Thus more than one cluster per output bit can be selected in the analysis to serve as an input to the clustering algorithm. To select certain clusters, criteria to make a choice are needed.

An input support set is worse than another one if it would, if implemented, result in a more
complex machine. According to (20) the complexity of a single cluster can be expressed in a quality factor $q_j$. This quality factor can be used to select a number of clusters to be passed on to the beam search algorithm. The lower $q_j$ is, better the analyzed cluster is.

### 4.3. Building the decomposition

Building the decomposition is the step in which a set of clusters is converted into a decomposed machine. The decomposed machine can still have symbolic states, but it must be clear from which partial machine(s) the returned state partitions $\tau_{sj}$ are obtained. It is the first step to state assignment. Making the connections between the partial machines doesn’t determine the state assignment, but it does put some boundary conditions on the state assignment.

Since in many practical cases the number of partial machines in a decomposition is relatively small and the number of connections between them is minimised, the number of possible solutions to the connection problem is small in comparison to the number of possible decompositions. Therefore an exhaustive search can be used to determine the optimal solution.

Demands for the returned state partition $\tau_{sj}$ are given in pairs of states $(s,t)$ which are not allowed to be in one block of $\tau_{sj}$. These pairs are stored in sets $T_j$. The exhaustive algorithm consists of three steps.

In the first step, for each pair $(s,t)$ of each set $T_j$ all possible pairs of state blocks from other partial machines which can realise this pair are searched and stored. These pairs of state blocks $B_k$, $B_i$ meet the demands:

$$\forall (s, t) \in T_j \exists i \neq j \exists B_{ik} \subset \pi^S_i \land B_{ij} \subset \pi^S_j \land s \in B_{ik} \land t \in B_{ij} \ .$$

If there is a pair $(s, t)$ which has no block pair $B_{ik}$, $B_{ij}$ it is not possible to make the connection and the clustering algorithm has computed an unimplementable decomposition. At the end of this section I will return to this problem.

In the second step each possible solution for each returned state partition $\tau_{sj}$ is considered. From the list of pairs of state blocks $B_i$, $B_j$ all combinations of pairs of blocks are chosen which realise $\tau_{sj}$ with a minimal number of bits. Combinations of the state block pairs which do this are saved and will be called a possible solutions for $\tau_{sj}$.

The third and last step in the algorithm uses the result of the second step. This result was a number of solutions for the connection problem in which all returned state partitions $\tau_{sj}$ can be realised in a minimal number of bits. In the third step, one solution from these solutions is chosen. This is a solution in which all state partitions $\pi^S_{sj}$ can be realised using a minimal number of bits.

Having found the optimal way to connect the partial machines the decomposition is complete and
can be saved.

If an unimplementable decomposition was encountered, it means that one or more returned state demands could not be resolved because some state information is not computed in the decomposed machine. With the $M_{in}$ and $M_{out}$ operators the input bits and old state information needed to compute the missing state information can be computed. Using this information the partial machine which can best be adjusted to compute the missing state information can be found. If there is no suitable partial machine to implement the missing state information in, even an extra partial machine, which then is a state machine computing the missing state information, can be defined. Thus the unresolved state demands can be resolved.

This solution to adjust partial machines to resolve unresolved returned state demands has not yet been implemented.

4.4. Don't care conditions

In the previous sections, the algorithms were described for decomposing completely specified sequential machines. Sequential machines however, can have don't care conditions which, if treated wrong, can lead to suboptimal or incorrect decompositions and which, if treated right, can be used to optimise the machine.

Looking at a next state and output table of a Moore machine, for the example in Table 1, three fields in which don't care conditions can occur can be distinguished. First, input bits in the input field can have don't care values. Second, certain output bits can have don't care values for certain present states. Entries in the next state field can have don't care values as well.

Don't care values for input bits can be dealt with radically by substituting all possible values for all don't care input bits and filling the next state table with the appropriate values. A machine with $m$ input bits would in this case always have $2^m$ columns in its next state table. A more subtle solution is to compute all mutually exclusive input bit patterns.

A set of mutually exclusive input bit patterns contains bit patterns which, if all possible values for all don't care bits are substituted don't give the same bit pattern for any two bit patterns from the set. The mutually exclusive set which covers all possible input bit patterns can now be used to identify all input bit patterns.

For example, consider a machine with three possible input bit patterns: --0, -01 and 1--. The set of mutually exclusive input bit patterns, covering all possible input bit patterns, contains the patterns

<table>
<thead>
<tr>
<th>$X_1X_2X_3$</th>
<th>00-</th>
<th>001</th>
<th>101</th>
<th>1-0</th>
<th>111</th>
<th>$Y_1Y_2Y_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>*</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>*</td>
<td>-1</td>
</tr>
</tbody>
</table>
Thus, creating one column for each mutually exclusive input bit pattern, not all the possible input bit patterns have to be stored. However, all input bit patterns which can occur are still contained in the set of mutually exclusive input bit patterns and correspond with exactly one column in the next state table.

Output bits can have don't care values for certain present states. Output bit y₂ in Table 1 for example, has a don't care value for present state 3. If Mₜ₋₀B(π₀Oₜ) is computed, with π₀Oₜ = \{y₂; y₁, y₃, y₄\} this results in two possible state partitions: \{(1, 4);(2, 3)\} or \{(1, 3, 4);(2)\}, depending upon how the don't care output bit value is assigned.

This Mₜ₋₀B partition is used in two situations in the decomposition algorithm. It is used in determining S_p (see (31)) and in computing the demands for τₜ₋₀ (see (23)). When using it for computing S_p, the don't care conditions can be satisfied by not considering the corresponding states (in the previous example state 3) while comparing state partitions.

When using the output bit values for computing τₜ₋₀, the don't care output bit values and the don't care next state entries have to be treated at the same time. Using the definition for the M-partitions, the product Mₜ₋₀B(π₀Oₜ)·Mₜ₋₅(π₅) can be computed without any problem under don't care conditions. Since after this product of M-partitions is computed the only operation that is performed on it is comparing it with other partitions, don't care conditions are allowed.

The last problem with don't care conditions will arise when mₜ₋₅ partitions should be computed. In the algorithm, the sum of π₅ₜ(π₅) partitions is used to ensure the existence of an IB-S partition pair (see (22)). Using don't care conditions in next state table entries, the equation

\[ m(π₁B₁ + π₁B₂) = m(π₁B₁) + m(π₁B₂) \]  \hspace{1cm} (38)

no longer holds. So instead of computing mₜ₋₅ partitions by adding π₅ₜ(π₅) partitions the mₜ₋₅ partitions have to be computed directly from the definition of the m-partition.

With the computation rules described above don't care conditions are allowed in the input field, output field and next state field of the next state and output table.
5. Implementation

The algorithms described in chapter 4 have been implemented in the form of a pascal program called dev.pas. This chapter will give information about the program structure and its use. The first part, about using the program is meant for people who want to work with the program and the second part, about the structure, for those who want to develop the program or use some of its routines from the program. Additional, more exhaustive software documentation is given in appendices 3 and 4.

5.1. Using DEV

5.1.1. System requirements

DEV is written in Domain Pascal version 8.8. So far, it has only been run under Domain/OS SR10.3 in a BSD 4.3 environment. Using extensions from the domain pascal has been avoided as much as possible, so the program should run using other pascal compilers without too many adjustments.

5.1.2. Installation and compilation

5.1.2.1. Directory structure

The directory structure that is constructed after installation is given in Figure 8. The directory called dev can be in the user directory, but if the default configuration file name is changed, it can be anywhere and have any name.

5.1.2.2. Files

Before installation all files listed in Text 1 have to be copied from the master copy to the directory source. These files are the source code of the program, a copy of a default configuration file and a file containing information for installing and compiling the program. The source code of the program is formed by the file dev.pas (the main program), the files ending on .mod (the modules of the program) and the files ending on .inc (the include files). The information for compiling and installing the program, used by the make utility, is in the file Makefile. All these
files will stay in the directory source.

Installing the program will cause the default configuration file `dev.config` to be copied to the directory `lib`.

### 5.1.2.3. Installation

To install the program, make a directory `source` and copy all files listed in Text 1 to this directory. The rest of the directory structure is created by the `make` utility.

To create this directory structure, compile the programs and copy all necessary files to the correct directories, go to the directory `source` and type `make install`.

### 5.1.2.4. Compilation

If the program has been installed compilation is not necessary; `make install` has already compiled
the program. If a user has made changes in the source code, however, the program needs to be recompiled. Recompiling is done by changing to the directory 'source' and typing 'make dev'.

5.1.3. File formats

The program DEV uses two types of files as input files: The input file containing the definition of the machine to be decomposed and the configuration file, containing a number of algorithm parameters. It can create four sorts of output files. The format of these files can be distinguished using their extension: .kis, .con, .clusters and .testdeftable. Examples of each type of file can be found in the example run in appendix 2. What information which file contains exactly is explained in chapter 5.1.4, in this section only the file formats are described.

All input and output files have a couple of common characteristics. All of them are ASCII files, so no characters which are not in the ASCII set can occur. Also, all files must end on a <return>. A <return> must be the last entry before an <end of file>.

The input file containing the machine to be decomposed is in the KISS file format. In this format '#' or '" as a first character of a line indicates that this line is a comment line. Apart from the comment lines, the input file must contain three other fields in the following sequence: a header, field with product terms and a field indicating the end of the file.

In the header, a number of machine parameters are defined: the number of states of the machine, the number of input bits to the machine and the number of output bits of the machine. Lines belonging to the header have '.' as the first character. The next characters are 's' or 'S' for indicating that the number of states is defined in this line, 'i' or 'I' for indicating that the number of input bits is defined and 'o' or 'O' to define the number of output bits. 'p' or 'P' can be the second character as well, to indicate the number of product terms of the machine, but it is ignored in the input file. These characters may also be separated by spaces or tabs from the leading '.'. After the dot and character a integer number must follow, for indicating the value of the machine parameter concerned. No other characters than mentioned before may occur in a header line.

The product terms consist of an input bit pattern, a present state, a next state and an output bit pattern. These three product term elements must be delimited by spaces or tabs and must be on one line.

The input bit pattern contains for each input bit '0', for indicating that the input bit is 0, '1', for indicating that it is 1 or '-', for indicating that it is don't care.

The present state and the next state fields can contain a state name or a '*'. The state name should consist of letters, digits and underscores; '*' indicates that the state is don't care.

The output bit pattern contains for each output bit '0', for indicating that the output bit is 0, '1', for indicating that it is 1 or '-', for indicating that it is don't care.

The field indicating the end of the file consists of '.e' or '.E'.

The output files with extension .kis contain definitions of partial machines in an adjusted
KISS-format. The entries in the product term field are adjusted to contain more input information. A line with a product term now consists of an input bit pattern, a returned state partition block number, a present state, a next state and an output bit pattern, all delimited by spaces. The bit patterns are the same as in the KISS format, but state names only contain digits and a returned state partition block number is added. The returned state partition block number is represented by a number of digits.

The configuration file contains comment lines, integer values and real values. The comment lines must start with '"' or '#'. The integer and real values are values for certain algorithm parameters. Since the program reads the values in a certain order, the order of these values may not be changed. One line may not contain more than one integer or real value. The integer values consist of digits, the real values are composed of digits and not more than one '.' or ',' for indicating the decimal point.

Output files with the extension .con contain comment lines and pairs of numbers. The comment lines again have a "" or '#' as their first character. The pairs of numbers are on one line and the two numbers are delimited by spaces or tabs. The first number indicates a tau block number and the second number a state block number. Both numbers are integer values composed of only digits.

The output files with the extensions .cluster and .testdeftable were not meant to be used by other programs, but the information stored there can be read by users. Their exact format can be found in the program source code. Both routines which write the files are in the module 'test.mod', .cluster is written by the procedure TestClusters and .testdeftable is written by the procedure TestDefTable.

5.1.4. Running DEV

Running DEV is described in this section and illustrated by an example run in appendix 2. For running DEV, one needs to have two input files: One input file in KISS format containing the definition of the sequential Moore machine one wants to decompose and a configuration file. The program DEV is in the directory dev and can be executed from this directory by typing 'dev'. When the program is started the user is first asked for the name (and if necessary the path) of the input file. After this filename is entered, the file is read and the user is asked for the name of the configuration file. If the default file name is used, a <return> can be given on this prompt. After this initial dialogue with the user, the program computes the decomposition of the machine defined in the input file using the algorithm parameters from the configuration file. If no fatal errors occur, a number of output files are written to the default directory. The name of the output files is created by deleting the extension and the path name from the input file and adding an extension to it. If the input file name is 'input/test.kis', for example, all output file names start with 'test' and have different extensions.
If it is specified in the configuration file, a file containing the next state table and output table is created. This file has the extension '.testdeftable' (in our example, the name would be 'test.testdeftable').

For each run of DEV, a number of decompositions can be saved. How many are to be saved can be specified in the configuration file. If n decompositions are saved, decomposition number i is described by the files from which the extension starts with '.i'.

For decomposition number i, a file with the extension '.i.cluster' is saved, if this was specified in the configuration file. It contains a description of the decomposition number i in partition notation. From this file one can read which state partition and which output partition is realised in which partial machine and which input partition and which state demands are needed in which partial machine.

If after decomposing a machine the connections between the partial machines can be created, for each partial machine j from decomposition number i two output files are written. These output files have the extensions '.i.j.kis' and '.i.j.con'.

The description of partial machine number j from decomposition number i in the modified KISS format is given in the output file with extension '.i.j.kis'. This file contains the number of states, input bits and output bits of partial machine j in its header and all product terms in the product term field. Product terms give the next state and output pattern for a certain input bit pattern, a certain present state and a certain returned state block number. The returned state block number is used to indicate how the partial machines are connected and in the file with the extension '.i.j.con' one can see which returned state block number belongs to which states in other partial machines.

In the comment lines of the output file with extension '.i.j.kis' the information is given which input bits and which output bits are active in partial machine j from decomposition i.

Together with each file with extension '.i.j.kis' an output file with extension '.i.j.con' is created to indicate how partial machine j is connected with other partial machines. This file contains a number of block pairs, each pair consisting of a returned state block number and a state block number. If block pairs 1 23 and 1 8 for example exist, this means that the partial machines should be connected in such a way that partial machine j is in returned state block 1 if and only if the partial machine containing in state 23 is in this state and the partial machine containing state 8 is in state 8.

To summarise which files can be created, we will give an example. Suppose a machine 'test.kis' is decomposed, resulting in three decompositions. The first decomposition has three partial machines and the third has two. In the second decomposition the connections could not be made. The output files that are created are:
5.1.5. Changing the constants

If machines input to DEV are large, it may happen that certain data structures which are used in DEV are not large enough. If this is the case, DEV will report an error which can be solved by changing a global constant in the source code. Constants which can be changed by the user are: MaxStateRange, MaxOutputBitRange, MaxInputBitRange, MaxInputPairPositionRange and MaxOutputPairPositionRange. The program will indicate in its error message which constants need to be changed. Note that the memory requirements of DEV can get extremely large when these constants are chosen too large and the performance can drop dramatically.

Note that if MaxInputBitRange or MaxOutputBitRange is changed, MaxWordLength might have to be changed as well. MaxWordLength must always have the maximum value of MaxInputBitRange and MaxOutputBitRange.

The constants that can be changed are in the include file 'locdecs.inc'. If any value has been changed the program has to be recompiled (with 'make dev').

5.2. The source code

In this section the structure of the source code of DEV is explained. A program is a sequence of operations on certain data, computing the required output data from the input data. Therefore this section will first describe how the most important data are stored and then it will give an impression of the program flow, which determines which operations in which order change the data.

5.2.1. Data structures

The main data that need to be stored consists of the machine definition and the sets of clusters. The machine definition is given in the form of a next state table and an output table. The next state table entries are chosen as a function of input bit patterns and present states, the output table
entries as a function of the present state. This can be quite easily implemented using arrays, so we will first consider the data structure for the cluster sets.

The cluster sets consist of clusters, each cluster representing one future partial machine and the information contained in a cluster can be represented by partitions. Partitions that need to be stored are:

- $\pi_{ib}$: An input bit partition indicating which input bits are used in the cluster.
- $\pi_{ob}$: An output bit partition indicating which output bits are computed in the cluster.
- $\pi_s$: A state partition containing the state information computed and stored in the cluster.
- $\tau_s$: A state partition containing the state information that is needed from other clusters to compute $\pi_{ob}$ and $\pi_s$.

Operations on the input bit partitions and output bit partitions are adding bits in important blocks, adding bits in the don’t care block and checking whether bits are in important or don’t care blocks. The structure that is suited best for these kind of operations is a set structure; if a bit is important in a bit partition, it is an element of the corresponding set, if bits are don’t care, they are not an element of the corresponding set.

So the structure used for input bit partitions and output bit partitions is:

- $InputBitPartition = \text{set of InputBitSeries;}$
- $OutputBitPartition = \text{set of OuputBitSeries;}$

with $InputBitSeries = \text{1..MaxInputBitRange}$ and $OutputBitSeries = \text{1..MaxOutputBitRange}$.

The operations on $\pi_s$ partitions are merging blocks, separating blocks, counting the number of blocks and checking whether two states are in the same block. These operations can all be implemented easily if a state partition is defined as

- $StatePartition = \text{array [StateSeries] of NextStateSeries;}$

with $StateSeries = \text{1..MaxStateRange}$ and $NextStateSeries = \text{0..MaxStateRange}$.

The index of a state partition thus indicates a state $s$ and the field it points to indicates the block state $s$ is in. The state partition $\{(1,4,6);(2,5);(3)\}$ thus is stored as the array with elements $1\ 2\ 3\ 1\ 2\ 1$. NextStateSeries contains the element 0 to be able to indicate that a state is don’t care in a partition, i.e. it can belong to any block of the partition.

In computing the partition $\tau_s$, not the entire partition needs to be computed. $\tau_s$ only needs to fulfill certain state demands and it can be stored as a set of state pairs $(s,t)$ which are not allowed to be in the same block of $\tau_s$. Since it is unknown exactly how many state pairs need to be stored for each partition $\tau_s$, it can best be stored in a dynamic list pointed to by a StateDemandPtr.

37
\[ StateDemandPtr = \text{^StateDemand}; \]
\[ StateDemand = \text{record} \]
\[ \quad s,t : \text{StateSeries}; \]
\[ \quad \text{next} : \text{StateDemandPtr}; \]
\[ \text{end}; \]

Now, a structure for a cluster set can be defined. A cluster set contains a number of clusters, which can best be stored in a list, because the number of clusters is unknown. Next to the four partitions mentioned above, a cluster must also contain a boolean variable \text{same}, to indicate that it contains the same output partition as the previous cluster (see section 4.2.7). The structure for a list of clusters:

\[ ClusterPtr = \text{^Cluster}; \]
\[ Cluster = \text{record} \]
\[ \quad \text{inpart} : \text{InputBitPartition}; \]
\[ \quad \text{outpart} : \text{OutputBitPartition}; \]
\[ \quad \text{pi} : \text{StatePartition}; \]
\[ \quad \text{tau} : \text{StateDemandPtr}; \]
\[ \quad \text{modtau} : \text{StateSeries}; \]
\[ \quad \text{same} : \text{boolean}; \]
\[ \quad \text{next} : \text{ClusterPtr}; \]
\[ \text{end}; \]

Next to a cluster list there are three more variables which need to be stored with each cluster set. These are \text{numclusters} indicating the number of clusters in the cluster set, \text{quality} to store the accumulated correlation factors (see 4.1.6) and \text{alternative1} to indicate whether the correlation factors are computed with alternative 1 or alternative 2. The structure for a list of cluster sets is:

\[ ClusterSetPtr = \text{^ClusterSet}; \]
\[ ClusterSet = \text{record} \]
\[ \quad \text{cluster} : \text{ClusterPtr}; \]
\[ \quad \text{next} : \text{ClusterSetPtr}; \]
\[ \quad \text{numclusters} : \text{integer}; \]
\[ \quad \text{quality} : \text{real}; \]
\[ \quad \text{alternative1} : \text{boolean}; \]
\[ \text{end}; \]

These are the most important data structures. A complete list of the structures used in the program is contained in the source code in the file 'loedecs.inc'.

5.2.2. Program flow

The program flow will be given in pseudo pascal. From the most important routines the structure
will be illustrated with comment and an indication which procedures or functions are called. In this section, only a limited number of routines is listed. For a more complete view see appendix 4.

The main program is called dev.pas, all routines are called from this program. For information on which module contains which routines, we refer to appendix 3 where the include files for all modules are listed.
5.2.2.1. Dev.pas

Program dev;
begin
  { initialise global variables }
  init_globals_adj;

  { read sequential machine inputfile }
  GetFSM;

  { read the configuration file }
  GetParameters;

  { construct the next state and output table }
  MakeNextStateTable;

  { copy the contents of the next state and output table to a testfile, if specified in the configuration file }
  if TestDefinitionTable then
    TestDefTable;

  { analyse all outputs }
  AnalyseOutput;

  { decompose the machine into clusters using the beam search algorithm }
  symbolicDecompositionBeam;

  { convert the clusters to a decomposition by connecting them and write the output files }
  decluster;
end. { dev }

5.2.2.2. AnalyseOutput

procedure AnalyseOutput;
{ Analysis the output bits one by one.
  Input : NextStTable and OTable containing the machine definitions.
  Output : MOutput, minput containing M S-OB and m IB-S partitions.
            Startclusterset containing one cluster set with for each cluster one active output.
}
begin
  { initialise m_IB-S and M_S-S partitions with one bit don't care resp. important }
  InitialiseMandMpartitions;

  { reserve memory for startclusterset and initialise this memory }

  { analyse all outputs }
  for OutputBitNum := 1 to OutputBitRange do
    AnalyseSingleOutput (OutputBitNum,StartClusterset)
end; { AnalyseOutput }

40
5.2.2.3. AnalyseSingleOutput

```pascal
procedure AnalyseSingleOutput ( y : OutputBitSeries; 
    var resultcluster : ClustersetPtr);

{ This procedure analyzes output bit y and constructs one or several 
  clusters with y the only active output bit from that cluster. }

Input: 
  y : Output bit
  mInput: m_IB-S partitions of input bit partitions with 
          one dontcare input bit.
  MOutput: M_S-OB partitions of output partitions with one 
           important output bit.
  NextStTable and OutTable for the definition of the machine.
Output: 
  resultcluster : The cluster(s) of the found possible partial machine(s 
                  are prepended to this list.
}

var 
  InputBit : InputBitRecord;          { containing the input bit(s) to be 
                                         deleted from the input support set }
  DefInputPart, NotRemovableInput : InputBitPartition; { essential input bits are active }
  DefOutputPart : OutputBitPartition;  { output partition with y active }
  DefStatePart, ZeroStatePart, auxpart : StatePart; 
  DefTauPart : StateDemandPtr;        { the state info that can be computed if input info=0 }
  cset, taumod : FullyConnectedSet;   { of DefTauPart }
  QTable : InputPairReduction;        { containing pairwise demands }
  QArray : QualHyarray;               { containing summed quality }
  chosen, nomorepos : boolean;        { should the constructed cluster be saved? }
  i : InputBitSeries;                 { no more input bits can be deleted }

poscluster : ClusterPtr;             { list of clusters which are likely to be reasonably good }

begin { AnalyseSingleOutput }

  { initialise output and input partition, from input partition compute state partition and 
    returned state demands } 
  InitialisePartitions (y, DefOutputPart, ZeroStatePart, DefInputPart, 
                       NotRemovableInput, DefStatePart, DefTauPart, cset, taumod);
  poscluster:=nil;

end; 
```

41
repeat

{ compute quality of making all pairs of important input bits don't care }
computeQtable(QTable, DefInputPart, NotRemovableInput,
DefOutputPart, DefStatePart, ZeroStatePart, taumod, y);

{ sum qualities for pairs to obtain qualities for single input bits }
sumQtable(DefInputPart, NotRemovableInput, Qarray, Qtable);

{ select the input bit(s) to be made don't care }
selectinputbits(DefInputPart, NotRemovableInput, Qarray, Qtable, InputBit);

{ select all constructed clusters, final selection in selectcluster }
chosen := (InputBit.length > 0);

if chosen then { save the found cluster as a possible result }
   savecluster(DefInputPart, DefOutputPart, DefStatePart,
   DefTauPart, cset, taumod, true, poscluster);

{ make input bits don't care and adjust state partition and state demands }
deleteinputbits(DefInputPart, InputBit, DefTauPart, cset,
DefStatePart, DefOutputPart, taumod);

{ Test whether the input support set is empty; if it is, quit }
nomorepos := true;
for i := 1 to InputBitRange do
   if NotRemovableInput[i] = notactive then
      nomorepos := nomorepos and (DefInputPart[i] = notactive);
until nomorepos;

{ put last cluster on poscluster list and return dynamic memory in DefTauPart }
savecluster(DefInputPart, DefOutputPart, DefStatePart,
DefTauPart, cset, taumod, true, poscluster);

{ return dynamic memory }
ClearStateDemand(DefTauPart);
clearcset(cset);

{ select a number of clusters to be returned and prepends these to resultcluster }
selectclusters(poscluster, resultcluster)
end; { AnalyseSingleOutput }
5.2.2.4. Symbolic decomposition beam

procedure symbolicdecompositionbeam ;
{
Uses a beam search algorithm to find a near-optimal solution for the symbolic
decomposition problem. StartClusterSet should contain at least one cluster set
to start the beam search with and the resulting cluster sets are stored in
completedClusterSet.

Input:
The global algorithm variables.
The machine definitions in NextStTable, InputMatrixDef and MOutput.

Modified:
StartClusterSet Containing one or more cluster sets. Each cluster set in the list
should contain at least one cluster.
After the routine is called StartClusterSet is empty.
CompletedClusterSet Containing the resulting cluster sets.
}
begin
numcandidateclustersets := 0;
if completedcluster = nil then
numcompletedclustersets := 0;
repeat
currentclusterset := StartClusterSet;
while currentclusterset <> nil do
begin
{ compute correlations between pairs of clusters from currentclusterset }
CalcCorrelationFactors ( currentclusterset, correlation, numpossiblepairs);

{ select a number of cluster pairs with high correlation }
DetermineCandidateClusterPairs ( correlation, candidatepair, numcandidatepairs,
numpossiblepairs, correlationfinalset);

{ should any more clusters be merged? }
if finalstate ( correlationfinalset, numcandidatepairs, currentclusterset)
then
{ this cluster set might be returned }
movefromstartlisttocompletedlist (currentclusterset)
else
begin
{ merge selected cluster pairs and store cluster set
in candidate cluster set list }
mergecandidatepairs ( candidatepair, numcandidatepairs, correlation,
currentclusterset, numcandidateclustersets);

currentclusterset := currentclusterset^next
end
end;
end;
copycandidatesettolist (numcandidateclustersets);
{ until no more cluster sets have clusters to be merged }
until startcluster = nil;

{ select the best cluster sets and return these in CompletedClusterSet }
selectresult
end; { symbolicdecompositionbeam }
5.2.2.5. Decluster

procedure decluster;
{
Declustering creates a decomposition of a set of clusters. All clusters per cluster set are taken and
the connections between the clusters are made. The algorithm that is used here is an exhaustive
algorithm; all possibilities are computed and the solution having the lowest number of demands for
the state assignment is chosen and written to the output files.

Input:  Completedclusterset.

Output:  Is written to the output files starting with ResultsFileName.
}
begin
 currentclusterset:=Completedclusterset;
 setcounter:=0;
 nosolutionpossible:=true;
 while currentclusterset<>nil do
 begin
  setcounter:=setcounter+1;
  if SaveClusters then
   outputclusters (currentclusterset, setcounter);
  CalcAllBlockDemands (currentclusterset, T);
  FindMinimalRealisations (currentclusterset, T, solutionpossible);
  if solutionpossible then
   nosolutionpossible:=false;
  FindMinimalStateAssignmentDemands (currentclusterset, T, S);
  OutputKissFormatAndConnection (currentclusterset, T, setcounter)
 end;
 currentclusterset:=currentclusterset^.next
 end;
 if nosolutionpossible then
 begin
  writeln ('None of the cluster sets found by the decomposition had a');
  writeln ('solution in which the connections could be made.');
  writeln ('You can try again with a broader beam search and saving');
  writeln ('more clusters.');
 end
end; { decluster  }
5.3. Sources of the source code

Not the whole source code has been written anew, some modules have been copied from the state assignment program maxad and some of them have been modified. The modules hash.mod, vector.mod and tree.mod have been copied without modification. The modules ainput.mod, atools.mod and NStable.mod have been copied and modified. The larger part of the source code, consisting of the main program dev.pas and the modules analysis.mod, beamdec.mod, decluster.mod, parttools.mod and test.mod have been written anew.
6. Decomposition example

In this chapter the algorithm is illustrated with an example. The example machine used is the sequential machine test.kis. The machine test.kis is the shift register shown in Figure 9. It has a parallel load of input bits X1, X2 and X3 enabled by input bit X4. Next to the output of the shift register, the output is formed by three output bits Y4, Y5 and Y6 which are the exclusive or of two state bits. The machine is represented in KISS format, having symbolic states and input and output bits in the file test.kis, listed in appendix 2. This machine will be decomposed in this chapter, taking the KISS table as an input to the algorithm. The algorithm should find a decomposition representing the structure in Figure 9, because this is an optimal structure. After the analysis of the output bits and the decomposition the result of the decomposition will be discussed.

6.1. Analysis

The analysis of the output bits will be illustrated by analyzing output bit Y4. Output bit Y4 has the primary input support set with elements X1, X3 and X4. With these input bits a 4-block state partition can be computed: \[ \pi^s = \{(1, 3);(2, 4);(5, 7);(6, 8)\} \]. A 2-block returned state partition is needed to form the necessary S-S and S-OB partition pairs.

To obtain a reduced input support set, input bits have to be deleted from the primary input support set. Deleting pairs of input bits and computing the state information that can be obtained without these input bits results in an identity state partition for all possibilities. Since no decision can be taken which input bit should be deleted from this, deleting one input bit from the primary support set is used to find an optimal input support set to start the decomposition with.

Deleting input bit X1 from the primary input support set results in the reduced input support set with elements X3 and X4. The state information that can be computed from this input support set is given by the 2-block state partition \[ \{\{(1, 3, 5, 7);(2, 4, 6, 8)\} \]. A 4-block returned state partition
is needed to form the necessary S-S and S-OB partition pairs. Deleting input bit X3 from the primary input support set results in the reduced input support set with elements X1 and X4. The state information that can be computed from this input support set is given by the 2-block state partition \{(1, 2, 3, 4); (5, 6, 7, 8)\}. A 2-block returned state partition is needed to form the necessary S-S and S-OB partition pairs. Deleting input bit X4 from the primary input support set results in the reduced input support set with elements X1 and X3. The state information that can be computed from this input support set is given by the identity state partition \{(1, 2, 3, 4, 5, 6, 7, 8)\}. A 2-block returned state partition is needed to form the necessary S-S and S-OB partition pairs. The input bit which results in the smallest number of blocks of the returned state partition when deleted is most likely to form a good cluster when deleted from the input support set. The two bits with the smallest number of blocks in the returned state partition are bits X3 and X4. Deleting bit X4 results in an identity state partition and the other input bit can be deleted from this input support set as well; no input information is needed to compute the identity state partition. A cluster containing an output bit and no state information or input information cannot be used in the clustering algorithm, because it will receive all its information from other clusters and contain no information itself. So, the best reduced input support set having two input bits is the set containing input bits X1 and X4.

Trying to find a better reduced input support set more bits must be deleted from the set containing input bits X1 and X4. Deleting either input bit, however, results in an identity state partition, which, as explained above, cannot be used in the beam search algorithm.

The optimal input support set for output bit Y4 is found. For other output bits optimal reduced input support sets can be found using the algorithm as well.

### 6.2. Clustering

The input to the clustering algorithm is formed by a cluster set containing 6 clusters. Each cluster contains one output bit, with the input and state information and the state demands calculated in the analysis. Whether clusters are merged or not depends on the correlation between these clusters. To illustrate the correlation, examples of high and low correlation will be given. After this, the merging tree, describing all the actions of the clustering algorithm, will be shown.

Three clusters from the start cluster set are (the returned state demands are ignored for a clear overview of the example):
output bit  \( Y_1 \)
input bits  \( X_1, X_4 \)
state partition \( \{(1, 2, 3, 4);(5, 6, 7, 8)\} \)

output bit  \( Y_4 \)
input bits  \( X_1, X_4 \)
state partition \( \{(1, 2, 3, 4);(5, 6, 7, 8)\} \)

output bit  \( Y_5 \)
input bits  \( X_2, X_4 \)
state partition \( \{(1, 2, 5, 6);(3, 4, 7, 8)\} \)

The cluster with output bit \( Y_1 \) and the cluster with output bit \( Y_4 \) have a high correlation. Both their input support sets and their state partitions are the same, so merging these two clusters would result in a cluster with a 2-bits input support set and a 2-block state partition.

The cluster with output bit \( Y_1 \) has a low correlation with the cluster with output bit \( Y_5 \). Merging these two clusters would lead to a cluster with a 3-bit input support set and a 4-block state partition.

Using this principle for the correlation factors between clusters, trying to estimate the relative increase of complexity when two clusters are merged, a beam search can be started, as illustrated in Figure 10.

At the top of the merging tree, an output partition \( 1 2 3 4 5 6 \) can be seen. Since the number on position \( j \) of this partition represents the cluster number output bit \( Y_j \) is in, all output bits are in separate clusters at the top of the merging tree.

An arrow represents a merge of two clusters with high correlation. As can be seen from the merging tree, not all possible merges are performed. With this merging tree, a maximum value of three merges per cluster set was defined.

A cluster set can exit from the merging tree in two possible ways. It can be deleted (indicated by the circles with a cross in Figure 10) or it can be moved to the list of completed sets (indicated by the rectangles). A cluster set can be deleted when more cluster sets with the same output partition are found in the merging tree. It is moved to the list of completed sets when little information is left to merge any more clusters. This is the case when all correlation factors are low.

When all cluster sets are either completed or deleted, the beam search algorithm is finished. The best cluster set is chosen from the list of completed sets and the rest is deleted. In the example the cluster set with output bit partition \( 1 2 3 1 2 3 \) (three clusters, cluster 1 containing output bits \( Y_1 \) and \( Y_4 \), cluster 2 output bits \( Y_2 \) and \( Y_5 \) and cluster 3 output bits \( Y_3 \) and \( Y_6 \)) is selected as the best cluster from the list of completed sets.

49
Building the decomposition means that the connections between the clusters in the cluster set selected by the clustering algorithm are created. This is done using an exhaustive search.

In the example machine test.kis, the input to this part of the algorithm was a cluster set containing three clusters. After the connections have been created between the clusters in this cluster set, the decomposition into three partial machines was found, as shown in Figure 11.

Since the structure of this resulting machine is the same as the structure of the machine test.kis (see Figure 9), the algorithm has found the correct decomposition.

Figure 10: The merging tree.

Figure 11: Decomposed machine test.kis.
7. Preliminary results

The developed software has been extensively tested when using a benchmark of various sequential machines in order to show that the algorithm computed valid decompositions and the program DEV indeed implemented the algorithm. All discovered bugs in the program have been removed and no more bugs have been discovered by further testing.

Some sequential machines have been decomposed using DEV to get an impression of the algorithm's performance. This performance has been checked using one criterium: If a machine can be split into a number of largely independent partial machines, the algorithm has to be able to find this decomposition, i.e. to discover the machine's internal structure.

Since no decomposition results for sequential machines were to our disposal with which the results could be compared, we have constructed some sequential machines with known optimal decompositions. A number of other sequential machines have been decomposed using an exhaustive search to get the material for comparison.

First the machine test.kis was decomposed. The results from this test can be found in the example run in appendix 2 and are illustrated in chapter 6. Test.kis is a machine which represents a shift register and some extra output bits in it. This structure was discovered by the algorithm using a very narrow beam.

A number of counters were also decomposed: a four bit Gray code counter, a four bit binary counter and a BCD counter.

Using a very narrow beam search with the counters, no other structure was found than implementing each output bit in one partial machine, using four partial machines.

An exhaustive search resulted in other decompositions with less than four partial machines. These decompositions had partial machines consisting of two or three merged partial machines from the solution with four partial machines as is illustrated in Figure 12. The decomposition into four partial machines results in $M_1$, $M_2$, $M_3$ and $M_4$; the decomposition into two partial machines in $M_1'$ and $M_2'$. Since $M_1'$ is composed of the connection of $M_1$ and $M_2$ and $M_2'$ of $M_3$ and $M_4$, both decompositions represent the same machine structure.

Some sequential machines have but one output bit, like the machine ex4.as we tried to decompose. Since the algorithm decomposes machines by creating an output partition, state machines or machines with only one, or few output bits cannot be decomposed.

When decomposing very large machines another problem arose. For example, scf.kis is a machine with 56 output bits, 27 input bits and 121 states. The program used too much memory and executed for too long to complete the decomposition. The analysis and the clustering algorithm
were completed in a reasonable time, but the building of the decomposition took too long. An exhaustive search for this part of the algorithm appears to be too complex for extremely large sequential machines. Developing heuristics for this part of the algorithm should solve this problem. These heuristics can, for example, rely on the fact that returned state demands which are obtained from one single partial machine are more likely to use a minimum number of state bits than returned state demands obtained from two or three partial machines. Also state blocks which are used more than once will have a higher probability to lead to a (near-) optimal solution.

Other machines, such as ex6.as, gave serial decompositions as a result of both the limited beam search and the exhaustive search. Parallel decompositions were also found for example ex5.as.

Summarising, in many cases the algorithm was able to discover the natural internal structure of a sequential machine; however it should be stressed that the only structure which can be found by the algorithm must be present in the sequential machine. Since not all machines have good decompositions, for the less structured machines the algorithm is unable to construct high quality decompositions because they do not exist.

The presented test results are only preliminary, and they give only an impression about the performance of DEV. The results are very promising, but more exhaustive testing is necessary to get more complete and more reliable information.
8. Conclusions

The aim of the reported research was to develop, implement and test an efficient heuristic method for the general simultaneous full decomposition of Moore machines with encoded inputs and outputs and symbolic states. It is required that the method will be able to discover the natural decompositional structure of the sequential machine, i.e. it will be able to find near optimal decompositions into largely independent partial machines which are sufficiently less complex than the original machine (if such decompositions exist for a certain machine).

The method developed constructs the limited set of near optimal decompositions, using a beam search bottom-up hierarchical clustering algorithm, with decision making based on correlations between the information flows in the sequential machine. For modelling the information flows and performing computations on them it uses the partition and partition pair algebras. Since the decomposition process performed by the method is based on constructing the appropriate output partitions, the method cannot be used for decomposing machines without or with only a few output bits.

It is quite easy to extend the method in order to decompose Mealy machines. The method has been implemented in the form of a Pascal program called DEY.

In order to get an impression about the performance of the method, a number of sequential machines has been decomposed using DEY. Some of these machines were constructed especially to perform the tests on (machines with known optimal decompositions, such as counters or shift registers); other machines were taken from the international benchmark set [18]. In all the checked cases (except for the case of machine ex4: the machine with a single output bit), the method was able to discover, by a very limited search, the natural decompositional structure of the sequential machine, i.e. to find the optimal simultaneous general full decomposition.

Among others, the method discovered the implementation structures typical for a modified shift register, a Gray code counter, a binary counter and a BCD counter. Also special cases of a general decomposition have been found by the method, such as a parallel decomposition (for ex5) and a serial decomposition (for ex6).

From this we can conclude that the preliminary results are very promising; however checking the method on a larger sample of sequential machines is necessary to obtain more complete and more reliable information about its performance.

Heuristic algorithms for multi-way general decomposition of sequential machines have also been the subject of the research work of S. Devadas et al. [13, 14, 15, 16, 17]. His algorithms, however, are related to the two-level symbolic logic minimisation. They are based on different heuristic cost functions which try to reflect in various ways the cost of a two-level logic implementation and not on the information flows in a sequential machine, as it is in our method.
Literature

[1] Hou, Y.
Trinity algebra and full decomposition of sequential machines.

Trinity algebra and its application to machine decompositions.

Symbolic analysis of a decomposition of information processing.

Algebraic structure theory of sequential machines.

The full decomposition of sequential machines with the state and output behavior realisation.
EUT Report 88-E-188.

The full decomposition of sequential machines with output behavior realisation.
EUT Report 88-E-199.

The full decomposition of sequential machines with the separate realisation of next state and output functions.
EUT Report 89-E-222.

[8] Jóźwiak, L.
The bit full decomposition of sequential machines.
EUT Report 89-E-223.

Bit full decomposition of sequential machines, algorithms and results.

[10] Jóźwiak, L.
Simultaneous decomposition of sequential machines.
An efficient method for the sequential decomposition of sequential machines.

An efficient method for decomposition of multiple-output boolean functions and assigned sequential machines.

Decomposition and factorization of sequential finite state machines.

[14] Devadas, S.
General decomposition of sequential machines: Relationship to state assignment.

Performance oriented decomposition of sequential circuits.

A unified approach to the decomposition and re-decomposition of sequential machines.

Optimum and heuristic algorithms for an approach to finite state machine decomposition.

[18] Lisanke, R.
Logic synthesis and optimizations benchmarks, version 2.0.

Algorithms for clustering data.
Appendices

Appendix 1: Configuration file

Appendix 2: Example run
Input file: test.kis
Test definition table: test.deftable
Cluster file: test.1.clusters
Partial machine definitions: test.1.1.kis
test.1.2.kis
test.1.3.kis

Input file: test.kis
Test definition table: test.deftable
Cluster file: test.1.clusters
Partial machine definitions: test.1.1.kis
test.1.2.kis
test.1.3.kis

Appendix 3: Include files

Appendix 4: Pseudo pascal description of the most important routines
dev.pas
ainput.mod
NStable.mod
analysis.mod
beamdec.mod
decluster.mod
parttools.mod
test.mod
Appendix 1: Configuration file

This appendix contains the default configuration file.
this is the configuration file for DEV.

It contains all algorithm parameters and these can be changed in this file.

For the program to be able to read all parameters, there are a few
values that must be obeyed.

First, comment can be put in the file, as long as lines containing comment
start with a double quote: ".

Second the order of the different parameters may not be altered. Otherwise
"wrong values are assigned to parameters.

Parameters can have integer values or real values. An integer value should
contain only digits from 0 to 9. No spaces, dots or anything else.

A real value can contain digits and dots or commas. Valid real values are:
1.2, 1.5, 0.5 etc.

After the value for the last parameter, a newline must always follow.

Below are the values for all parameters.

TestDefinitionTable
Indicates whether the definition table is written to the output. If
TestDefinitionTable is 0 no definition table output is created, otherwise
it is created.

MaxNumClustersPerOutput
This is an integer value indicating how many clusters one output can start
the beam search with.

F
Together with the following two values, a1 and a2, this value determines
how the correlation factors are calculated.
"The higher F is, the more important the number of state bits for the
complexity of a submachine is.

a1
This value determines how important the number of connections of the
decomposed machine is for the total complexity.

a2
This value determines how important the complexity of submachines of the
decomposed machine is for the total complexity.

MaxNumCandidateClusterpairs
Together with the next parameter the width of the beam search is determined
by this one. This much candidate cluster pairs are maximally selected to be
merged.

MinCorrelationBetweenClusters
Together with the previous parameter the width of the beam search is
"determined by this one. Candidate cluster pairs are only selected to be
merged if the correlation between the clusters is higher than this value.

MinNumClustersPerSet
"and
MaxNumClustersPerSet
"Solutions found will consist of k clusters, with
"MinNumClustersPerSet <= k <= MaxNumClustersPerSet

SameCorrelationDeviation
"and
MinNumSameCorrelationClusters
"A cluster set is said to be a good solution, and saved, if
"MinNumSameCorrelationClusters cluster pairs which have the highest correlation
"have a deviation in correlation of maximal SameCorrelationDeviation.

MaxNumStartClusterSets
This parameter is to limit the beam from getting too wide. The next step in the
beam search is not started with more than MaxNumStartClusterSets cluster sets.
"RelativeReductionClusterSets"
"If there are more than MaxNumStartClusterSets candidate cluster sets, only a
"fraction RelativeReductionClusterSets is copied to start the next step in the
"beam search with.
0.25

"MaxNumSavedFinalClusterSets"
"Indicates how many cluster sets may be returned from the beam search.
1

"SaveClusters"
"Indicates whether cluster sets are saved in the partition notation. If SaveClusters
"is 0 only KISS-output and connection information output is created, otherwise
"clusters are also saved in the partition notation.
1

"This line is inserted to ensure that there is a newline after the last parameter
"value. DO NOT DELETE THESE LINES."
Appendix 2: Example run

This appendix contains an example run of DEV. The input and output files are listed. As configuration file the default configuration file from appendix 1 was used.

Input file: test.kis
Test definition table: test.deftable
Cluster file: test.1.clusters
Partial machine definitions: test.1.1.kis, test.1.1.con, test.1.2.kis, test.1.2.con, test.1.3.kis, test.1.3.con
.14
.o6
.6#

0001 * 1 -------
0011 * 2 -------
0101 * 3 -------
0111 * 4 -------
1001 * 5 -------
1011 * 6 -------
1101 * 7 -------
1111 * 8 -------
--- 0 1 1 000111
--- 0 2 5 001010
--- 0 3 2 010100
--- 0 4 6 011001
--- 0 5 3 100001
--- 0 6 7 101100
--- 0 7 4 110010
--- 0 8 111111

."
The next state and output definition table
derived from the input file
//dev/uk/test.kin

Input definition array:
1 0001
2 0011
3 0101
4 0111
5 1001
6 1011
7 1101
8 ----0
9 1111

Output definition array:
1 ----
2 000111
3 001010
4 010100
5 011001
6 100001
7 101100
8 110010
9 111111

State definition names
state names are used when states are printed
state numbers give the positions of names in partitions

State | State name
1 | 1
2 | 2
3 | 3
4 | 4
5 | 5
6 | 6
7 | 7
8 | 8

Output table
State | Output
1 | 2
2 | 3
3 | 4
4 | 5
5 | 6
6 | 7
7 | 8
8 | 9

Next state table
State | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
1 | 1 | 1 | 1 | 6 | 3 | 5 | 7 | 9 | 2
2 | 1 | 4 | 2 | 3 | 1 | 6 | 5 | 9 | 7
3 | 1 | 7 | 4 | 9 | 6 | 1 | 6 | 5 | 7
4 | 1 | 8 | 3 | 1 | 7 | 3 | 4 | 5 | 9
5 | 1 | 9 | 2 | 4 | 5 | 7 | 6 | 8 | 9

-----------------------------------
Cluster number 1
InputPartition 0 0 1 1
Outputpartition 0 0 1 0 1
Statepartition 1 2 1 2 1 2 1 2
modulus tau 2

tau:
{ 6, 8 }
{ 5, 7 }
{ 4, 6 }
{ 3, 5 }
{ 2, 8 }
{ 2, 4 }
{ 1, 7 }
{ 1, 3 }

Cluster number 2
InputPartition 0 1 0 1
Outputpartition 0 1 0 1 0
Statepartition 1 1 2 2 1 1 2 2
modulus tau 2

tau:
{ 4, 8 }
{ 4, 7 }
{ 3, 0 }
{ 3, 7 }
{ 2, 6 }
{ 2, 5 }
{ 1, 6 }
{ 1, 5 }

Cluster number 3
InputPartition 1 0 0 1
Outputpartition 1 0 0 1 0 0
Statepartition 1 1 1 1 2 2 2 2
modulus tau 2

tau:
{ 7, 8 }
{ 6, 7 }
{ 5, 0 }
{ 5, 6 }
{ 4, 0 }
{ 2, 3 }
{ 1, 4 }
{ 1, 2 }

66
<table>
<thead>
<tr>
<th>active input bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>active output bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 6</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1 01</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>2 01</td>
</tr>
<tr>
<td>-0</td>
<td>1</td>
<td>1</td>
<td>1 01</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>2</td>
<td>1 10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>2 10</td>
</tr>
<tr>
<td>-0</td>
<td>1</td>
<td>2</td>
<td>1 10</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>1</td>
<td>1 00</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>2 00</td>
</tr>
<tr>
<td>-0</td>
<td>2</td>
<td>1</td>
<td>2 00</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>2</td>
<td>1 11</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>2 11</td>
</tr>
<tr>
<td>-0</td>
<td>2</td>
<td>2</td>
<td>2 11</td>
</tr>
</tbody>
</table>
This file contains information to connect clusters to a decomposed machine. For each tau-block in the second column of the KISS-output, the state demands for this block is listed.

```
# taublock    stateblock
  1        3
  2        4
```

END OF FILE
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>3</td>
<td>3 01</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>3</td>
<td>4 01</td>
</tr>
<tr>
<td>-0</td>
<td>1</td>
<td>3</td>
<td>3 01</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4</td>
<td>3 10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>4</td>
<td>4 10</td>
</tr>
<tr>
<td>-0</td>
<td>2</td>
<td>3</td>
<td>3 00</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>3</td>
<td>3 00</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>3</td>
<td>4 00</td>
</tr>
<tr>
<td>-0</td>
<td>2</td>
<td>4</td>
<td>4 11</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>4</td>
<td>3 11</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>4</td>
<td>4 11</td>
</tr>
<tr>
<td>-0</td>
<td>3</td>
<td>4</td>
<td>4 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This file contains information to connect clusters to a decomposed machine. For each tau-block in the second column of the KISS-output, the state demands for this block is listed.

<table>
<thead>
<tr>
<th>tau-block</th>
<th>state-block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

# END OF FILE
# active input bits:
# 14

<table>
<thead>
<tr>
<th>Active output bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01  1</td>
</tr>
<tr>
<td>11  1</td>
</tr>
<tr>
<td>-0  1</td>
</tr>
<tr>
<td>01  2</td>
</tr>
<tr>
<td>11  2</td>
</tr>
<tr>
<td>-0  2</td>
</tr>
<tr>
<td>01  1</td>
</tr>
<tr>
<td>11  1</td>
</tr>
<tr>
<td>-0  1</td>
</tr>
<tr>
<td>01  2</td>
</tr>
<tr>
<td>11  2</td>
</tr>
<tr>
<td>-0  2</td>
</tr>
</tbody>
</table>
This file contains information to connect clusters to a decomposed machine. For each tau-block in the second column of the KISS-output, the state demands for this block is listed.

<table>
<thead>
<tr>
<th>taublock</th>
<th>stateblock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

END OF FILE
Appendix 3: Include files
HStable.inc :
procedure MakeNextStateTable; extern;

input.inc:
procedure GetFSM; extern;
procedure GetParameters; extern;
procedure AnalyzeOutput; extern;

procedure savecluster (var DefInputPart : InputBitPartition;
var DefOutputPart : OutputBitPartition;
var DefStatePart : StatePartition;
var DefStatePart : StateDemandPtr;
var saved : StateSeries;
samein : boolean;
var poscluster : ClusterPtr); extern;

tools.inc :
procedure initGlobals_adj; extern;
procedure program_exit; extern;
procedure Check_10Status; extern;

function PowerOf2( A: NonNegativeInteger): integer32; extern;
procedure GetUnsignedValue(var value: integer); extern;
procedure DecodeInPairPosition : InputPairSeries;
var FirstPattern, SecondPattern : InputSeries; extern;
procedure OrderStates( var A, B : StateSeries); extern;
procedure SwapStates( var A, B : StateSeries); extern;
procedure Locate(FirstState, SecondState: StateSeries;
var StatePairPosition : StatePairSeries); extern;
procedure Decode(StatePairPosition: StatePairSeries;
var FirstState, SecondState: StateSeries); extern;

procedure QuickSort( L : StatePairSeries;
var ItemVectorCounter: VectorStatePairPosition; extern;
procedure clean_up(); extern;

beamdecl.inc :
procedure symbolicDecompositionbeam; extern;

decluster.inc:
procedure decluster(); extern;

hash.inc :
procedure InitTable; var T : NameTable; var ItemsSoFar : integer; extern;

function Hash (var Name : workString; NameLen : integer): integer; extern;
function Equal (var A : workString; Alen : integer;
var B : workString; Blen : integer); boolean; extern;
procedure SearchAndInsert(var T : NameTable;
  var itemName : vrrString;
  itemNameLen : integer;
  var itemNum : integer;
  var itemsSoFar : integer ); extern;

parttools.inc : 
procedure ClearStateDemand (var TDem : StateDemandPtr); extern;
function StatePartSum (var P1,P2 : StatePartition) : StatePartition; extern;
function TauPartMult (var P1,P2 : StatePartition) : StatePartition; extern;
function StatePartMult (var P1,P2 : StatePartition) : StatePartition; extern;
function InputPartMult (var I1,I2 :InputBitPartition) : InputBitPartition; extern;
function OutputPartMult (var O1,O2 :OutputBitPartition) :OutputBitPartition; extern;
function InputPartSum (var I1,I2 :InputBitPartition) :InputBitPartition; extern;
function OutputPartSum (var O1,O2 :OutputBitPartition) :OutputBitPartition; extern;
function StatePartitionSmallerOrEqual (var P1, P2 :StatePartition) : Boolean; extern;
function MSOB (y : OutputBitSeries) : StatePartition; extern;
function mISS (var Ipart : InputBitPartition) : StatePartition; extern;
function hSS (var SPart : StatePartition) : StatePartition; extern;
procedure clearcset (var cset : FullyConnectedSet); extern;
procedure findfullyconnectedsubsets (TauPtr var eset : StateDemandPtr;
  FullyConnectedSet); extern;
function StateDemandModule (var cset : StateSeries); extern;
function HodulusStatePart (var SPart : StatePartition); extern;
function HodulusInputPart (var I : InputBitPartition); extern;
function HodulusOutputPart (var O : OutputBitPartition); extern;
function ModSumStateDemands (var cset1, cset2 : FullyConnectedSet); extern;
function ModMultStateDemands (var cset1, cset2 : FullyConnectedSet); extern;
function InputPatternDefer (var I1,I2 : VectorSet;
  var IPart : InputBitPartition) : Boolean; extern;

75
tree.inc:

function CompareVecs(var W1, W2 : VectorInputDef) : integer; extern;
procedure InsertToTree(var Root : InputPattTree; var Patt : VectorInputDef); extern;
procedure ExtractRoot(var Root : InputPattTree; var Patt : VectorInputDef); extern;
procedure PrependToInpList(var InpList : InputNodePtr; var IW : VectorInputDef); extern;
procedure Merge(var Tree1 : InputPattTree; var Chain : InputNodePtr); extern;
procedure SwapTrees(var Tree1, Tree2 : InputPattTree); extern;
procedure ClearInpList(var List : InputNodePtr); extern;
procedure MergeLists(var tolist : InputNodePtr;
  var fromlist : InputNodePtr); extern;

vector.inc:

function Intersection(A, B : VectorInputDef) : InputNodePtr; extern;
procedure SplitInputVectors(var InpTree : InputPattTree;
  var ResultList : InputNodePtr); extern;
procedure ComplementISpace(var inplist : InputNodePtr;
  var complist : InputNodePtr); extern;
Appendix 4:
Pseudo pascal description of the most important routines

All routines are sorted per cluster that they are in. To determine which routine is in which module use the rule that unless a procedure or function is in an include file listed in appendix 3, the routine is in the same module it is called from.

The modules listed in this appendix are:

dev.pas
ainput.mod
NStable.mod
analysis.mod
beamdec.mod
deccluster.mod
parttools.mod
test.mod
Program dev;

begin

  { initialise global variables }
  init_globals_adj;

  { read sequential machine input file }
  GetFSM;

  { read the configuration file }
  GetParameters;

  { construct the next state and output table }
  MakeNextStateTable;

  { copy the contents of the next state and output table to a test file, if stated in the configuration file. }
  if TestDefinitionTable then
    TestDefTable;
  else
    AnalyseOutput;

  { decompose the machine into clusters using the beam search algorithm }
  symbolicdecompositionbeam;

  { make the clusters to a decomposition by connecting them and write the output files }
  decluster;

end. { dev }
procedure GetFSM;

(* This routine is responsible for processing the input data of the program *)
(* After initialization of the internal variables the procedure asks for the *)
(* name of the datafile. Then the data from the file "InpFile" is explored the *)
(* headers and the finite state machine definition. Basic internal structures *)
(* are filled, like "DefTable" and some important factors are computed, like *)
(* "StatePairPositionRange" and "PartitionRange". *)

begin

(* Initialise variables and structures *)
InitTable(StateNameTable, StatesSoFar);
InitTable(OutWordsTable, OutputPatternRange);

(* Read machine filename *)
(* Search filename without extension or pathname *)
(* Summary : outputfiles are in the default directory *)

(* processing inputfile *)

(* Skip commentlines in input file *)
(* get all header entries and initialise InputBitRange, StateRange and *)
(* OutputBitRange *)

(* "don't care" vector, i is a dummy *)
It gets output pattern number 011111

(* read FSM definition *)
repeat

(* read input pattern *)
(* read present state *)
(* read next state *)
(* read output pattern *)

next := DefTable[PresSt];
DefTable[PresSt] := next;
ProductTerms := succ(ProductTerms);
end;
NewLine;

(* Search for .e or end of file *)
if eof(InpFile) then
    finished:=true
else
    finished:= (InpFile = '.');
until finished;
close(InpFile);

if StatesSoFar <> StateRange then Error(Inc_St_Hum);

(* fill output patterns definition table *)
(* OutputPattern 0 is the dontcare pattern *)

(* Calculate machine parameters *)
StatePairPositionRange := StateRange * (StateRange - 1) div 2;
PartitionRange := 0;
while PowerOf2(PartitionRange) < StateRange do
    PartitionRange := succ(PartitionRange);
end; (* GetFSM *)

procedure GetParameters;

(* Reads algorithm parameters from the file with name ConfigFileName. *)
begin
(* open configuration file *)
(* Skip commentlines in inputfile *)
(* Read the algorithm parameters *)
(* read next real value from a text file. The real value should be *)
on one line and nothing else should be on that line. 0.5 can be represented by:
0.5 0.5 .5 or .5

) ReadIntParam:
reads next integer value from a text file. The integer value should be
on one line and nothing else should be on that line
)

ReadIntParam (InpFile, MaxNumClustersPerOutput);
ReadRealParam (InpFile, F);
ReadRealParam (InpFile, a1);
ReadRealParam (InpFile, a2);

ReadIntParam (InpFile, MaxNumCandidateClusterPairs);
ReadIntParam (InpFile, MinCorrelationBetweenClusters);

ReadIntParam (InpFile, MinNumClustersPerSet);
ReadIntParam (InpFile, MaxNumClustersPerSet);
ReadRealParam (InpFile, SameCorrelationDeviation);
ReadIntParam (InpFile, MinNumSameCorrelationClusters);

ReadIntParam (InpFile, MaxNumStartClusterSets);
ReadRealParam (InpFile, RelativeReductionClusterSets);

ReadIntParam (InpFile, MaxNumSavedFinalClusterSets);

ReadIntParam (InpFile, SaveClusterInt);
SaveClusters := SaveClusterInt <> 0;

close (InpFile)
end; (GetParameters )
module nextstatetabletools;

Procedure FillInputTable (InputList : InputNodePtr; var InputMatrixDef : DefArray); 
{
This procedure copies the input patterns from the InputList to the global input definition array InputMatrixDef. It also adjusts the global variables InputPatternRange and InputPairPositionRange.

Input List : Linked list of used input patterns.
Output InputMatrixDef : Array containing all input patterns.
InputPatternRange : The number of input patterns in InputMatrixDef
InputPairPositionRange : The number of possible combinations of two input patterns.
}
begin
InputPatternRange := 0;
Inputptr := Inputlist;
{ copy the input list to the input definition matrix }
while Inputptr <> nil do
begin
  if inputpatternrange = maxinputpatternrange then begin
    writeln('Value out of range in FillInputTable');
    writeln('Increase const MaxInputPatternRange');
    writeln('in the file decs.inc on recompile');
    program_exit;
  end;
  InputPatternRange := succ (InputPatternRange);
  InputMatrixDef [InputPatternRange] := InputPtr.inpWord;
  Inputptr := InputPtr.next;
end;

InputPairPositionRange := (InputPatternRange* (InputPatternRange - 1)) div 2;
end; {FillInputTable }

procedure FillOutputTableEntry ( Staterr : integer; var DefTab : FSMDefTable; var OTEntry : OTTableEntry); 
{ The output table is input independent for Mealy machines. This is checked while assigning the proper value to the OTEntry. }
Input : Staterr : The state entry for which the output table is filled in.
Output : OTEntry : The number of the output pattern for state Staterr.
}
begin
Defptr := DefTab [Staterr];
{ check if machine is a Moore machine }
while Defptr <> nil do
begin
  out := defptr.outnum;
  Defptr := defptr.next;
  if (Defptr <> nil) then if (Defptr.outnum <> out) then begin
    writeln('Warning: The input machine is not a Mealy machine.');
    writeln('Program will continue, taking the output last mentioned');
    writeln('with each state and treating the machine as a Moore');
    writeln('machine.');
  end;
end;

OTEntry := out;
end; { FillOutputTableEntry }

procedure FillNextStateTableEntry ( StateNr : integer; var DefTab : FSMDefTable; var InputMatrixDef : DefArray; var NStEntry : NStateEntry); 
{ This procedure fills a single entry of the nextstate table for *
{ a present state specified by the parameter StateNr. *
{ at Inputs. *
{ StateNr - number of a state the entry is filled for *
{ DefTab - finite state machine's [FSM] description table read from *
{ the input file *
{ InputMatrixDef
begin
  (* fill the entries which are listed in InputMatrixDef *)
  for InputNum := 1 to InputPatternRange do
    begin
      DefPtr := DefTab[StateNr];
      (* fill entries where input pattern is included *)
      (* (covered) into non-split input vectors list from "DefTab" *)
      (* Check case of don't care present-state *)
      (* In this case the don't care entry is 'expanded' *)
      (* i.e. inputpat - no outputpat *)
      (* will be included as *)
      (* inputpat <statenr> no outputpat *)
      (* if it is not covered yet it must be a real don't care entry *)
      if not(covered) then
        begin
          NSTEntry[InputNum] := 0;
        end;
    end;
  end;
end: (* FillDefTableEntry *);

procedure MakeNextStateTable;
{
  this procedure calculates the entire next state table and the output table.
  Inputs:
  DefTable        - FSM definition table
  InputMatrixDef  - Definition of the input patterns
  InputPatternRange and InputPairPositionRange
  NextStTable     - With split input patterns
  OutTable        - For a Mealy machine
}
begin
  { initialise the tree "T", the result- and newinputlists }
  T := nil;
  ResultList := nil;

  (* copy the input pattern of all states into the tree "T" *)
  (* except if nextstate and output are both don't care *)

  { split the tree T to obtain a list of mutually exclusive input patterns. }
  SplitInputVectors(T, ResultList);

  { fill the input definition table }
  FillInputTable ( ResultList, InputMatrixDef );

  { fill the next state table and the output table }
  for S:=1 to staterange do
    begin
      FillNextStateTableEntry ( S, DefTable, InputMatrixDef, NextStTable[S] );
      FillOutputTableEntry ( S, DefTable, OutTable [S] );
    end;

  ClearInpList(ResultList);
end: (* MakeNextStateTable *);
module analysis;
{
declare types which are only used in this module
}
Type
DelInputPair = Record
q : StateSeries;
end;

InputPairReduction = array [InputBitSeries,InputBitSeries] of DelInputPair;
{ containing quality of input bit pairs }
QualityArray = array [InputBitSeries] of Integer; { quality of input bits }

InputBitRecord = Record
body : array [InputBitSeries] of InputBitSeries;
{ containing input bits to be deleted }
length : InputBitSeries; { number of input bits to be deleted }
end;

procedure InitialisePartitions (y: OutputBitSeries);
var
DefOutputPart : OutputBitPartition;
ZeroStatePart : StatePartition;
DefInputPart : InputBitPartition;
NotRemovableInput : InputBitPartition;
DefTauPart : StateDemandPtr;
var set : FullyConnectedSet;
{ Calculates the primary input support belonging to output y. This is the input set giving enough state information to calculate output y. Also a number of other partitions are initialised
Input: MOutput and mInput: y
Output: DefOutputPart: The M S-ON and m IB-S partitions.
DefInputPart: The output partition.
NotRemovableInput: The input partition containing the primary input support.
ZeroStatePart: An input partition. Bits are active when deleting these input bits from the primary input support means that no state information can be calculated.
DefTauPart: The state partition that is the m_IB-S partition of the input bit partition without state information.
set For storing the state demands.
}

procedure CalculateQtable (var QTable : InputPairReduction;
var InputPart : InputBitPartition;
var HRInput : InputBitPartition;
var OutputPart : OutputBitPartition;
var ZeroStatePart : StatePartition;
tauMod : StateSeries);
{ The table which contains for each entry the quality of the cluster when a pair of input bits are deleted from the input support set is calculated.
Input : InputPart, HRInput, OutputPart, ZeroStatePart, mInput, MOutput and tauMod.
Input : (Table containing for each pair of input bits i, j) which are not active in HRInput and with k-<, the quality factor. If k is the HRInput partition can be adjusted as well, if the m-particle of the bit partition with bit i deleted results in ZeroStatePart.
}
begin
Dummy:=nil;
dummy:=nil;
for k:=1 to InputBitRange do
auxInputPart[k]:=InputPart[k];
for i:=1 to InputBitRange do
begin
  if (InputPart [i] = active) and (NRInput [i] notactive) then
  begin
    auxinppart [i]:=notactive;
  end;
  for j:=i to InputBitRange do
  begin
    if (InputPart [j] = active) and (NRInput [j] = notactive) then
    begin
      (calculate the state partition with deleted
       input bits i and j from the input partition)
      auxinppart [j]:=notactive;
      SPart:=mIBS (auxinppart);
      (calculate the returned state demands)
      CleCStateDemand (SPart, OutputPart, TDummy, caset, tmod);
      ClearStateDemand (TDummy);
      clearsct (cdummy);
      (save the quality and if i=j the other information)
      QTable [i,j].q := (tmod - taumod);
      if i=j then
      if StatePartitionSmallerOrEqual (ZeroStatePart, SPart) then
        NRInput [i]:=active
      else
        auxinppart [j]:=active
    end;
  end;
end; // CalculateQTable

procedure savecluster (var DefInputPart : InputBitPartition;
var DefOutputPart : OutputBitPartition;
var DefStatePart : StatePartition;
var DefTauPart : StateDemandPtr;
var caset : FullyConnectedSet;
taumod : StateSeries;
samein : boolean;
var poscluster : ClusterPtr);
{ Saves the data in DefInputPart, DefTauPart, DefStatePart and
  module au to form a cluster in the cluster list poscluster.
Input
  : DefInputPart, DefOutputPart, DefTauPart, caset, DefStatePart,
  taumod and samein.
Modifies
  : prepends one entry to the list poscluster. }

procedure sumQtable (var QArray : QualityArray;
var QTable : InputPairReduction);
{ The quality table is summed to obtain the quality array.
Input
  : QTable, DefInputPart, NotRemovableInput.
Modifies
  : QArray }
begin
  i:=1;
  while i < InputBitRange do
  begin
    if (DefInputPart [i] active) and (NotRemovableInput [i] notactive) then
    begin
      QArray [i]:=0;
    end;
  end;
I

File iualys.csmt Created at 2:00am on Tuesday, January 1, 1960

vandijk Page 3 of 6

j:=1;
while j<InputBitRange do
begin
  { j an active bit ? }
  if (DefInputPart[j]=active) and (NotRemovableInput[j]=notactive) then
    if (i<=j) then
      QArray[i]:=QArray[i]+QTable[i,j].q
    else if (i>j) then
      QArray[i]:=QArray[i]+QTable[j,i].q;
  end
  j:=j+1
end

(QArray); SomeQtable ];

{******************************************************************************}

procedure selectinputbits ( var
  DefInputPart, NotRemovableInput : InputBitPartition;
  var
  QArray, QTable, InputBit : QualityArray;
  var
  InputBitPartitation, InputPairReduction, InputBitRecord );

Searches in QArray and QTable for the input for which the consequences of deleting it
are smallest, i.e. with minimum QTable [i] or, if those are equal minimum
QTable [i,i].q. If there are more input bits with minimum quality all of those are
saved in InputBit.

Input
  : QArray, QTable, DefInputPart, NotRemovableInput.
Modifies
  : InputBit.
}

{******************************************************************************}

procedure deleteinputbits (var
  DefInputPart, InputBitPartitation : InputBitPartition;
  var
  InputBit, DefTauxPart : StateDemandPtr;
  var
  QArray, QTable, DefStatePart, DefOutputPart : QualityArray;
  var
  taumod : StateSeries);

{ Deletes one the input bit(s) in InputBit from DefInputPart and calculates the
state demands and returned state (tau) demands.

Input
  : DefInputPart, DefOutputPart, QTable, InputBit.
Modifies
  : DefInputPart, DefTauxPart, QTable, DefStatePart, taumod.
  These form a new cluster complying with the demands.
}
begin
  { delete input bit(s) in InputBit from DefInputPart }
  { calculate state partition, tau demands }
  DefStatePart := mIBS (DefInputPart);
  ClearStateDemand (DefTauxPart);
  clearset (caset);
  CalcStateDemand (DefStatePart, DefOutputPart, DefTauxPart, caset, taumod)
end;

{******************************************************************************}

procedure selectclusters (var
  poscluster : ClusterPtr;
  var
  resultcluster : ClusterPtr );

{ Select maximal MaxNumClustersPerOutput clusters from the list poscluster.
Return these in resultcluster. The clusters are chosen on the basis of a quality
factor: the smaller the quality factor, the better the cluster.

Modified: resultcluster: The selected clusters are appended and numclusters is adjusted.
poscluster: The list of clusters from which clusters are chosen. This list

85
function qualitycluster (currentcluster : ClusterPtr): real;
{
  calculates the quality of currentcluster
}
begin
  with currentcluster do
  begin
    modinput := ModulusInputPart (inpart);
    tau1 := ln{cor*ln (modtau)} / 2;
    mmlmr := 2*(modinput + 4*taumod +
               (5*F)*ln{cor*ln (ModulusStatePart (pi))}/(11+F));
  end;
  qualitycluster := a1*numconnections + a2*complexsubmach
end;

begin
  first order the quality of all cluster sets contained in completed
  clusterset. The ordered qualities are stored in a quality list qlist.
}
(search which quality threshold should be taken to delete the required
number of clusters)
{ if there is more than one clusters with the threshold quality, save
all clusters with quality > threshold quality (even if more than
MaxNumClustersPerOutput are selected) }
{ dispose of memory used by qlist }  
{ prepend poscluster to startclusterlist. The first cluster will have same
  = false and the rest same=true; }  
end;

procedure AnalyseSingleOutput (y OutputBitSeries; var resultcluster : ClustersetPtr);
{
  This procedure analyses output bit y and constructs one or several
  clusters with y the only active output bit from that cluster.
}
Input:
y      : Output bit
Input : m_k-OB partitions of input bit partitions with
       one dontcare input bit.
Output : M_0-OB partitions of output partitions with one
         important output bit.
NextBitTable and OutTable for the definition of the machine.

Output:
resultcluster : The cluster(s) of the found possible submachines are prepended to this list.
}
var
  InputBit      : InputBitRecord;  { containing the input bit(s) to be
delated from the input support set }
  DefInputPart, NonRemovableInput
  DefOutputPart, DefStatePart,
  ZeroStatePart, support
  NextStatePart
  FullConnectedSet;  { of DefTupPart }
  StateSeries;  { module of returned state demands }
  InputFailsReduction;  { containing priliminary quality }
  QualArray;  { containing summed quality }
  keep, noInput : boolean;  { should the constructed cluster be saved? }
  poscluster : ClusterPtr;

procedure AnalyseSingleOutput;

begin ( AnalyseSingleOutput )

{ Initialise output and input partition, from input partition calculate state partition and returned state demands }
InitialisePartitions ( y, DefOutputPart, ZeroStatePart, DefInputPart, NotRemovableInput, DefStatePart, DefTauPart, cset, taumod );

poscluster:=nil;
repeat
{ calculate quality of making all pairs of important input bits don't care }
calculateQtable ( QTable, DefInputPart, NotRemovableInput, DefOutputPart, DefStatePart, ZeroStatePart, taumod, y );
{ sum qualities for pairs to obtain qualities for single input bits }
sumQtable ( DefInputPart, NotRemovableInput, Qarray, Qtable );
{ select the input bit(s) to be made don't care }
selectinputbits ( DefInputPart, NotRemovableInput, Qarray, Qtable, InputBit );
{ select all constructed clusters, final selection in selectcluster }
chosen := ( InputBit.length > 0 );
if chosen then { save the found cluster as a possible result }
savecluster ( DefInputPart, DefOutputPart, DefStatePart, DefTauPart, cset, taumod, true, poscluster );
{ make input bits don't care and adjust state partition and state demands }
deleteinputbits ( DefInputPart, InputBit, DefOutputPart, DefStatePart, DefTauPart, taumod );
{ Test weather the input support set is empty; if it is, quit }
nomorepos:=true;
for i:=1 to InputBitRange do
if NotRemovableInput [i]=notactive then
  nomorepos := nomorepos and ( DefInputPart [i] = notactive );
until nomorepos;
{ put last cluster on poscluster list and return dynamic memory in DefTauPart }
savecluster ( DefInputPart, DefOutputPart, DefStatePart, DefTauPart, cset, taumod, true, poscluster );
ClearStateDemand ( DefTauPart );
{ select a number of clusters to be returned and prepend these to resultcluster }
selectclusters ( poscluster, resultcluster );
end; { AnalyseSingleOutput }

{**************************************************************************}

procedure InitialyseManImpartitions;

{ given the machine in the next state and output table, calculate m_IR-S and M_S-OH partitions of resp. input partitions with input bit k in the don't care block and the rest of the bits important, and output partitions with output bit j important and the rest in the don't care block. }

Input: NextStateTable and OTable
Output: mInput and mOutput

begin
{ create input bit partition with dcb = w: }
mInput [k] := mIBS ( IPart );
for j:=1 to OutputBitRange do
  mOutput [j] := mOB ( j );
end; { InitialyseManImpartitions }

{**************************************************************************}

procedure AnalyseOutput;

{ Analyse the output bits one by one. }
Input: NextStTable and CTable containing the machine definitions.

Output: MOutput, mInput containing M S-OB and m IB-S partitions.

Start clusterset containing one cluster set with for each cluster one
active output.

begin

{ initialise m_IB-S and M_S-S partitions with one bit don't care resp. important }
InitialiseHandmpartitions;

{ reserve memory for startclusterset and initialise this memory }

{ analyse all outputs }
for OutputBitNum := 1 to OutputBitRange do
  AnalyseSingleOutput (OutputBitNum,StartClusterset)

end; { AnalyseOutput }
module head;

declare types which are only used in this module

Type

correlation array = array [OutputPairSeries] of real;
( for storing the correlation factors between a pair of clusters )
candidate pairs array = array [OutputPairSeries] of OutputPairSeries;
( for storing which cluster pairs are candidates )

*******************************************************************************

procedure CalcCorrelationFactors : current clusterset : Cluster setPtr;
var correlation : correlation array;
var num correlation pairs : OutputPairSeries;

Calculates the correlation between any two clusters in clusterset which are not in one group with same = true and stores it in correlation. Correlation is a one dimensional array. In correlation [1] the correlation between the first and second cluster is stored, in correlation [2] the correlation between the first and third etc.

Input: clusterset: The clusterset to be analysed.
the algorithm parameters F, a1, a2.

output: correlation: The array containing the correlation factors.

begin
cluster1: =currentclusterset".cluster;
correlationarray: = nil;
while (cluster1 <> nil) do begin
cluster2: = cluster1".next;
( skip clusters with cluster2".same = true, increasing correlationcounter )
if currentclusterset".alternative then
while (cluster2 <> nil) do begin
cluster2: =cluster2".next;
end;
cluster1: = cluster1".next
end;
num possible pairs: = correlationcounter
end; //CalcCorrelationFactors

*******************************************************************************

procedure DetermineCandidateClusterPairs
var correlation : correlation array;
var candidate pairs : candidate pairs array;
var num candidate pairs:
num possible pairs : OutputPairSeries;
var correlationfinalset: boolean;

Selects MaxNumCandidateClusterPairs candidate cluster pairs to be stored and stores those in candidate pairs. Cluster pairs with the highest correlation are selected and selected clusters have a minimum correlation of MinCorrelationBetweenClusters.
Further the boolean correlationfinalset is calculated. If this boolean is true,
the current cluster set will be moved to the list CompletedClusterSet. It is true if no candidate pairs are selected. Also if MinNumSameCorrelationDeviation cluster pairs with the highest correlation have a difference of less than SameCorrelationDeviation in correlation.

Input: correlation: The array containing the correlation factors for each cluster pair.
numpossiblepairs: The number of possible cluster pairs.

Output: candidatepair: Array containing the numbers of the selected candidate cluster pairs, ordered in decreasing correlation.
numcandidatepaires: The number of selected candidate cluster pairs.
correlationfinalset: Indicating whether the current cluster is a final cluster set based on the correlation factors.

val:
 clusterpairnum,
 l,
umselectedpairs, (number of cluster pair numbers stored in candidatepair)
prevel,prevel\ , OutputPairSeries;
currentcorrelation, (correlation of cluster pair number clusterpairnum)
firstcorrelation : real;

begin
 { order cluster pairs with a correlation higher than
  MinCorrelationBetweenClusters in decreasing order in the array
  candidatepair } 

determine how many cluster pairs are to be merged } 
if (numselectedpairs > MaxNumCandidateClusterPairs) then
 numselectedpairs := MaxNumCandidateClusterPairs 
else
 numselectedpairs := numselectedpairs;

 { calculate correlationfinalset } 
end; { DetermineCandidateClusterPairs } 

*******************************************************************************

function finalstate (correlationfinalset : boolean;
 numcandidatepaires : OutputPairSeries;
 curcluster : ClusterSetPtr) : boolean;

| Determines whether the current cluster set is used to merge more clusters or it is saved. If necessary the calculation of the correlation factors is also switched from alternative1 to alternative2. |

var numnotsameclusters : OutputBitSeries;
curcluster : ClusterPtr;

begin
 { count not same clusters = the number of different output bit partitions } 

if (correlationfinalset or (numcandidatepaires=0)) 
and (numnotsameclusters > MaxNumClustersPerSet) 
and curclusterset.alternative1 then

begin
 finalstate:=false;
curclusterset.alternative1:=false 
end

else

finalstate := correlationfinalset and
 (numnotsameclusters <= MaxNumClustersPerSet) and
 (numnotsameclusters <= MinNumClustersPerSet) or
 (numcandidatepaires = 0); (last line to prevent eternal loop ) 

end; { finalstate } 

*******************************************************************************

procedure movestartclustertocompletedlist (var clusterset : ClusterSetPtr); 

Searches cluster set in list Startclusterset and moves it to the list Completedclusterset. Numstartclustersets and numcompletedclustersets are adjusted.
procedure mergecandidatepairs (var candidatepair : candidatepairarray; 
numcandidatepairs : OutputPairSeries;
var correlation : correlationarray; 
var currentset : ClusterSetPtr; 
var numcandidateclustersets : integer); 
{ Merges the cluster pairs from clusterset indicated in candidatepair. The resulting cluster sets are prepended to the list candidateclusterset. 
Input: Candidatepair, numcandidatepairs: Indicating which cluster pairs should be merged. 
Clusterset: The cluster set from which clusters are merged. 
Correlation: Needed to calculate the quality of the new cluster sets. 
Output: Numcandidateclustersets and candidateclusterset. } 
begin 
for i=1 to numcandidatepairs do 
begin 
{ search cluster pair cluster1, cluster2 with number j which is to be merged } 
new (newclusterset); 
new (newclusterset".cluster); 
{ calculate the cluster consisting of the two merged clusters } 
with newclusterset".cluster" do 
begin 
inpart:=InputPartMult (cluster1".inpart, cluster2".inpart); 
outpart:=OutputPartMult (cluster1".outpart, cluster2".outpart); 
p:=mIBS (inpart); 
tau:=nil; 
CalcStateDemand (p, outpart, tau, modtau); 
same:=false; 
next:=nil 
end; 
newclusterset".numclusters := 1; 
{ copy the rest of the cluster set (excluded the two merged clusters and the rest of the groups containing the merged clusters with same = true) in the new cluster set and link this set to candidateclusterset. } 
{ then link the cluster set to the candidate cluster set } 
newclusterset".quality := currentset".quality 
+ correlation [candidatepair [i]]; 
newclusterset".alternative := currentset".alternative; 
newclusterset".next := candidateclusterset; 
candidateclusterset:=newclusterset; 
umcandidateclustersets := numcandidateclustersets+1; 
end 
end; end;  
{ mergecandidatepairs } 

*******************************************************************************
Then all remaining cluster sets in the list Candidateclusterset are moved to the list Startclusterset.

} function sameopart [var o1, o2 : outputpartition]: boolean; { returns true if o1 and o2 are the same. }
end; { sameopart }

begin
{ delete all startclustersets }
{ make a list containing the output partition of each clusterset }
{ look if clusters have the same output partition. If they have, delete them }
if numcandidateclustersets > MaxNumStartClusterSets then begin
{ delete a number of candidate cluster sets }
{ first order the quality of all cluster sets in a quality array qlist }
{ search which quality threshold should be taken to delete the required number of cluster sets }
{ if there are more than one cluster sets with the threshold quality, find the number of cluster sets with the threshold quality which should not be deleted }
{ Delete candidate cluster sets: cluster sets with a quality > minq are not deleted. Also numskipdelete cluster sets with a quality equal to minq are not deleted. The rest is deleted. }
{ dispose of memory used by the quality list qlist }
end;

{ move candidate clustersets to list Startclusterset }
Startclusterset := Candidateclusterset;
Candidateclusterset := nil;
numstartclustersets := numcandidateclustersets;
numcandidateclustersets := 0
end; { copycandidateclustersetostartset }

*******************************************************************************

function qualityclusterset (var currentset : ClustersetPtr): real;
{ Calculates the quality of the current cluster set. This quality is based on the number of connections between the submachines [dimension 1] and the complexity of each submachine formed by one cluster [dimension 2]. If a number of clusters is the same, i.e. have the same output, the best cluster is chosen and the rest deleted. }

Input: currentset : A list of cluster sets forming a cluster set. No clusters will be the same after the procedure is called.
        a1, a2, F : Algorithm parameters.

Output: qualityclusterset : The calculated quality of the currentset.
end; { qualityclusterset }

*******************************************************************************

procedure selectresult;
{ Procedure to select MaxNumSavedFinalClusterSets from the list Completedclusterset. To select sets a quality factor is calculated. The lower this quality, the better the cluster set is. }

Input: Completedclusterset: containing the clustersets.
        a1, a2, F and MaxNumSavedFinalClusterSets : algorithm parameters.

Output: Completedclusterset: containing the selected cluster sets.
        numcompletedclustersets: the number of clustersets in Completedclusterset
end;
{ first order the quality of all cluster sets in a quality list qlist }

92
search which quality threshold minq should be taken to delete the required number of cluster sets

if there are more than one cluster sets with the threshold quality, save all cluster sets with threshold quality (even if more than MaxNumSavedFinalClusterSets) are selected

Delete completed cluster sets: cluster sets with a quality <= minq are not deleted.

dispose of memory used by glist
end; ( selectresult )

**********************************************************************************************************************************************

procedure symbolicdecompositionbeam ;

Uses a beam search algorithm to find a near-optimal solution for the symbolic decomposition problem. StartClusterSet should contain at least one cluster set to start the beam search with and the resulting cluster sets are stored in completedClusterSet.

Input :
The global algorithm variables. The machine definitions in NextSTable, InputMatrixDef and MOutput.

Modified: StartClusterSet : Containing one or more cluster sets. Each cluster set in the list should contain at least one cluster. After the routine is called StartClusterSet is empty.
CompletedClusterSet : Containing the resulting cluster sets.

begin
numcandidategroupsets := 0;
if completedclusterset = nil then
numcompletedclustersets := 0;
repeat
currentclusterset := StartClusterSet;
while currentclusterset <> nil do
begin
{ calculate correlations between pairs of clusters from currentclusterset }
CalcCorrelationFactors ( currentclusterset, correlation, numpossiblepairs );

{ select a number of cluster pairs with high correlation }
DetermineCandidateClusterPairs ( correlation, candidategroup, numcandidategroups, numpossiblepairs, correlationfinalset );

{ should any more clusters be merged? }
if finalstate ( correlationfinalset, numcandidategroups, currentclusterset ) then
{ this cluster set might be returned }
movefromstartlisttocompletedlist ( currentclusterset );
else
begin
{ merge selected cluster pairs and store cluster set in candidate cluster set list }
mergecandidategroups ( candidategroup, numcandidategroups, correlation, currentclusterset, numcandidategroupsets );

currentclusterset := currentclusterset.next
end
end;
copycandidategroupsettolist ( numcandidategroupsets );
{ until no more cluster sets have clusters to be merged }
until startclusterset = nil;

93
{ select the best cluster sets and return these in CompletedClusterset }
selectresult
end;  { symbolizedCompositionBeam }
module decluster;

{ declare data types which are only used in this module }

type
  BlockdemandPtr = ^BlockDemand;
  BlockDemand = record
    k: 1; { the cluster number of the realising state partition }
    k: 1; { the realising state blocks }
    next : BlockdemandPtr;
  end;

  BlockdemandListPtr = ^BlockdemandList;
  BlockdemandList = record
    block : BlockDemandPtr; { blocks realising state pair }
    next : BlockdemandListPtr;
  end;

  BlockdemandListArray = array [OutputBitSeries] of BlockdemandList;

procedure CalcAllBlockDemands (var currentset : ClusterSetPtr;
 var T : BlockdemandListArray);

Each pair (s, t) from the state demand list tau from each cluster from
 currentset is treated. All pairs of blocks from all state partitions pi from
 currentset that contain enough information to calculate the pair (s, t) are
 stored in T.

function twolog (i : integer) : integer;

Returns the two logarithm of i as an integer. Twolog (i) = ln(i) / ln(2).

procedure FindMinimalRealisations (var currentset : ClusterSetPtr;
 var T : BlockdemandListArray;
 var solutionpossible : boolean);

From all possible state assignment demands stored in T, those demands
 are selected which have a realisation in a minimal number of bits. The selected
 demands are stored and returned in T. If absolutely no solution can be found,
 solutionpossible is false, otherwise true is returned.

begin
  clustercounter:=0;
  currentcluster:=currentset^.cluster;
  solutionpossible:=true;
  while (currentcluster<>nil) and solutionpossible do
  begin
    clustercounter:=clustercounter+1;

    ( allocate memory for curblocklist and minblocklist and
      initialize curblocklist for first combination of blockdemands)

    ( find minimal realisations for this T [clustercounter] )
    finished := (curblocklist = nil) or not solutionpossible;
    minnonrealisations := twolog (StateRange);
    while not finished do
    begin
      ( find number of bits of current realisation )
      foundbits := 0;
      for searchcounter:=1 to currentset^.numclusters do
      if searchcounter<>clustercounter then
      begin
        stateelem:=nil;
        searchblocklist:=Curblocklist;
        foundbits :=
      end
      if foundbits = minnonrealisations then
      begin
        ( store minimal realisation and update T )
        curblocklist := curblocklist^;
        curblocklist^.next := nil;
        curblocklist^.realisations :=
      end
      if solutionpossible then 
      begin
        solutionpossible := false;
        
      end
      if currentset^.statepartition then
      begin
        stateelem := stateelem^;
        if searchcounter<>clustercounter then
        begin
          stateelem := stateelem^;
          
        end
      end
    end
  end
end
while auxblocklist<>nil do
  begin
    if auxblocklist^.block^.j = searchcounter then
      begin
        new (auxstatedem);
        auxstatedem^.a:=auxblocklist^.block^.k;
        auxstatedem^.t:=auxblocklist^.block^.l;
        auxstatedem^.next:=statedem;
        statedem:=auxstatedem
      end;
      auxblocklist:=auxblocklist^.next;
    end;
  foundstatebits:=foundstatebits+twolog (StateDemandModulus [statedem]);
  ClearStateDemand [statedem]
end;

{ this one better than the previous ones ? }
{ yes: delete current minblocklist }

{ was it better than previous ones, or just as good as previous ones }
{ yes: add curblocklist to the minblocklist }

{ select next possible combination of blockdemands; }
{ store it in curblocklist. If no more combination: finished := true } end;

{ dispose of memory used by curblocklist }

{ copy minblocklist to T [clustercounter]. Dispose of other blockdemands }
  currentcluster:=currentcluster^.next
end;

{ FindMinimalRealisations }

******************************************************************************

procedure FindMinimalStateAssignmentDemands (var currentclusterset:ClustersetPtr; var T : Blockdemandlistarray; var S : StateDemandArray);

 begins

  ( Searches T to find an optimal realisation of the connections. A realisation is optimal when it can be realised with a minimal number of state bits and the constraints put on the state assignment is minimal.

  The optimal realisation found is stored in S and T: S contains for each state partition pi the blockdemands, i.e. pairs of blocks with numbers s and t which must be separated by one bit. T contains for each returned state partition tau the source for the state bits: the separation of states s and t is realised by separating blocks k and l from cluster j. )

begin

  ( initialize structure to store structure for current possibility for T; store it in curblocklist )

  ( and initialise curpi and minpi to store blockdemands for states )

  foundoptimal:=false;
  finished:=false;
  minequality:=OutputBitRange * StateRange;

  repeat

    ( calculate all block demands for all pi in all clusters. )
    curcluster:=currentclusterset^.cluster;
    clustercounter:=0;
    while curcluster<>nil do
      begin
        clustercounter:=clustercounter+1;
        auxblocklist:=curblocklist [clustercounter];
        while auxblocklist<>nil do
          begin
            statecluster:=auxblocklist^.block^.j;
            new (auxtau);
            auxtau^.o:=auxblocklist^.block^.k;
            auxtau^.t:=auxblocklist^.block^.l;
            auxtau^.next:=auxblocklist;
auxtau.next:=curpi [statecluster].
curpi [statecluster]:=auxtau;
auxblocklist:=auxblocklist.next
end;
curcluster:=curcluster.next
end;

{ calculate the quality of this realisation }
curcluster:=currentclusters.next.cluster;
clustercounter:=0;
quality:=0;
possibleoptimalrealisation:=true;
while curcluster<>nil do
begin
clustercounter:=clustercounter+1;
q:=StateDemandModulus (curpi [clustercounter]);
if twolog (q) > twolog (ModulusStatePart (curcluster.pi)) then
possibleoptimalrealisation:=false;
quality:=quality+q;
curcluster:=curcluster.next
end;

{ is this realisation better than previous ones? }
(minimal number of state bits possible and minimal number of demands)
if (quality < minquality) and
((possibleoptimalrealisation and foundoptimal) or
(not possibleoptimalrealisation and not foundoptimal)) or
(possibleoptimalrealisation and not foundoptimal)
then
begin
minquality:=quality;
foundoptimal:=possibleoptimalrealisation;
end;

{ save curpi in minpi and curblocklist in minblocklist }
end
else
{ dispose of memory in curpi }

{ select next possible curblocklist; }
if no more blocklists possible, finished := true
until finished;

{ dispose of memory used by different lists, copy minblocklist to T
and minpi to S }
end: { FindMinimalStateAssignmentDemands }

-********************************************************************

procedure itos ( i : integer;
var s : FileNameString );
{ calculates the string representing integer i and returns it in s }
-********************************************************************

procedure writeminputbitpattern ( var f : text;
var ipart : InputBitPartition;
inputnum : InputSeries );
{ Writes an input bit pattern to f. Only bits active in ipart are written. }
-********************************************************************

procedure writeoutputbitpattern ( var f : text;
var opart : OutputBitPartition;
outputnum : OutputSeries );
{ Writes an output bit pattern to f. Only bits active in opart are written. }
-********************************************************************

procedure writestateenum ( var f : text;
var pi : StatePartition;
s, stateoffset : StateSeries );
begin
  if a=0 then
    write (f, '
   )
  else
    write (f, pi [a] + stateoffset:10, '
);
end;

*******************************************************************************

procedure OutputKissFormatAndConnection (var currentclusterset : ClustersetPtr;
  var T : BlockdemandListArray;
  setcounter : integer);
{
  Outputs cluster set currentclusterset in two files per cluster; one for the
  definition of the submachine (having extension .kis) and one containing
  information to connect it with other submachines.
Input:
  currentclusterset: The set of clusters which form a machine.
  T: Containing information to connect the
  submachines.
  setcounter: The number of this cluster set.
}
begin
  curcluster:=currentclusterset^.cluster;
  clustercounter:=0;
  curstateoffset:=0;
  while curcluster<>nil do
  begin
    clustercounter:=clustercounter+1;
    open stream for KISS output (result) and for connections
    output (connection) ;
    ( calculate tau-partition of current cluster: tau })
    if (curcluster^.modtau > 1) then
      multpitau := StatePartMult (tau, curcluster^.pi)
    else
      multpitau := curcluster^.pi;
    ( write KISS output: )
    ( first write the header and comment lines containing information about
      which input and output bits are used in this submachine. )
    ( write all product terms; use multpitau InputPatternDefBr to write each
      product term exactly once )
    writeln (result,' .e');
    ( write connection information: )
    ( first write the header containing information in comment lines )
    ( then write pairs of tau- and state block numbers indicating which
      tau blocks are realised by which state blocks )
    writeln (connection, '.e');
    ClearStateDemand (tau, stateblocks);
    curstateoffset:=curstateoffset+ModulusStatePart (curcluster^.pi);
    curcluster:=curcluster^.next;
    close (result);
    close (connection);
  end;
end;

*******************************************************************************

*******************************************************************************

procedure outputclusters ( var currentset : ClustersetPtr;
  setcounter : integer);
{
  Write cluster set information to output file.
}
begin
  pathname:="' ;
  write (setcounter, setcounterstr);
  append (pathname, ResultfileName, '.', setcounterstr, '.clusters');
  open (result, pathname, 'UNKNOWN', IOstatus.all);
  Check_IOstatus;
end;
procedure decluster;

Declustering is a decomposition of a set of clusters. All clusters per cluster set are taken and the connections between the clusters are made. The algorithm that is used here is an exhaustive algorithm; all possibilities are calculated and the solution having the lowest number of demands for the state assignment is chosen and written to the output files.

Input: CompletedClusterset.
Output: Is written to the output files starting with ResultsFileName.

begin
    currentclusterset:=CompletedClusterset;
    setcounter:=0;
    nosolutionpossible:=true;
    while currentclusterset<>nil do
    begin
        setcounter:=setcounter+1;
        { save cluster set in partition notation if specified in configuration file }
        if SaveClusters then
            outputclusters (currentclusterset, setcounter);
        { look where all state pairs (s,t) can be solved, store it in T }
        CalclAllBlockDemands (currentclusterset, T);
        { look how all state pairs (s,t) in T can be solved with a minimal modulus of tau, store it in T }
        FindMinimalRealisations (currentclusterset, T, solutionpossible);
        if solutionpossible then
        begin
            nosolutionpossible:=false;
            { find the solution in T which uses the minimal number of limitations to the state assignment,
            making a minimal state modulus possible }
            FindMinimalStateAssignmentDemands (currentclusterset, T, S);
            { output this solution using various files }
            OutputKissFormatAndConnection (currentclusterset, T, setcounter);
        end;
        currentclusterset:=currentclusterset^.next
    end;
    if nosolutionpossible then
    begin
        writeln ('None of the cluster sets found by the decomposition had a solution in which the connections could be made.');
        writeln ('You can try again with a broader beam search and saying more clusters.');
    end;
end; { decluter     }
module partools;

{*******************************************************************************

procedure ClearStateDemand (var TDem : StateDemandPtr);
{ Return the memory used by the linked list TDem to the system, TDem is nil after the procedure is called }
}{*******************************************************************************

function InputPartMult (var I1,I2 : InputBitPartition):InputBitPartition;
{ Multiplies bit partitions I1 and I2. The resulting partition is returned. Neither I1 nor I2 is altered. }
}{*******************************************************************************

function ModMultiInputPart (var I1,I2 : InputBitPartition):InputBitSeries;
{ Multiplies bit partitions I1 and I2. The modulus of the resulting partition is returned. Neither I1 nor I2 is altered. }
}{*******************************************************************************

function OutputPartMult (var O1,O2 : OutputBitPartition):OutputBitPartition;
{ Multiplies bit partitions O1 and O2. The resulting partition is returned. Neither O1 nor O2 is altered. }
}{*******************************************************************************

function InputPartSum (var I1,I2 : InputBitPartition):InputBitPartition;
{ Sums bit partitions I1 and I2. The resulting partition is returned. Neither I1 nor I2 is altered. }
}{*******************************************************************************

function ModSumInputPart (var I1,I2 : InputBitPartition):InputBitSeries;
{ Sums bit partitions I1 and I2. The modulus of the resulting partition is returned. Neither I1 nor I2 is altered. }
}{*******************************************************************************

function OutputPartSum (var O1,O2 : OutputBitPartition):OutputBitPartition;
{ Sums bit partitions O1 and O2. The resulting partition is returned. Neither O1 nor O2 is altered. }
}{*******************************************************************************

function ModulusStatePart (var SPart: StatePartition): StateSeries;
{ returns the modulus of the state partition SPart, i.e. the maximal block number in this partition. SPart is not altered. }
}{*******************************************************************************

function ModulusInputPart (var I : InputBitPartition): InputBitSeries;
{ The number of active bits in partition I is returned. No data is altered. }
}{*******************************************************************************

function ModulusOutputPart (var O : OutputBitPartition): OutputBitSeries;
{ The number of active bits in partition O is returned. No data is altered. }
}{*******************************************************************************

function StatePartSum (var P1,P2 : StatePartition): StatePartition;
{ The sum of the symbolic state partitions P1 and P2 is returned. Neither P1 nor

100
P2 should contain don't care information. If they do, blocks of the resulting partition are not merged due to the don't care information. P1 nor P2 are altered.

```plaintext
function StatePartitionMult (var P1, P2 : StatePartition) : StatePartition;
{
    The multiplication of the symbolic state partitions P1 and P2 is returned.
    P1 nor P2 can contain don't care information. If a state is don't care in either partition, it will be in the multiplication as well. P1 nor P2 are altered.
}
```

```plaintext
function TauPartMult (var P1, P2 : StatePartition) : StatePartition;
{
    The multiplication of the symbolic state partitions P1 and P2 is returned.
    P1 and P2 can contain don't care information. All don't care states in one partition form one block, which can be intersected by blocks from the other partition.
    P1 nor P2 are altered.
}
```

```plaintext
function StatePartitionSmallerOrEqual (var P1, P2 : StatePartition) : boolean;
{
    Compares the partitions P1 and P2. If P1< P2 the returned value is true, otherwise it is false. P1 nor P2 is altered.
}
```

```plaintext
function MSOB (y : OutputBitSeries) : StatePartition;
{
    Calculates the M_S-O/B partition of an output bit partition, i.e. all the state information that is needed to calculate the output bit partition with all bits don't care, except bit y.
    Input: y
     OutputMatrixDef : the definition matrix of the output;
                      maps outputnums to output bit patterns.
     OutTable : a table containing the output number for a present state.
    Output: MSOB : the resulting state partition.
}
```

```plaintext
function InputPatternDefer (var I1, I2 : VectorDef;
                          var IPart : InputBitPartition) : boolean;
{
    Compares input vectors I1 and I2. Returns true if and only if the two vectors defer on a bit position which is 1 - not don't care for either input vector or 2 - active in Ipart.
    I1, I2 nor IPart are altered.
}
```

```plaintext
function mIBS (var IPart : InputBitPartition) : StatePartition;
{
    Calculates the m_IB-S partition of an input bit partition, i.e. all the state information that can be calculated from the input bit partition. The input partition is the partition with all bits important, except bit 1, which is don't care. The m_IB-S partition is calculated starting with a zero state partition and merging state blocks if necessary.
    Input: I
     InputMatrixDef : the definition matrix of the input;
                      maps inputnums to input bit patterns.
     NextSTable : a table containing the next state as a function of inputnum and present state.
    Output: mIBS : the resulting state partition.
}
```
function CalcNSOB_SSpart (var pi : StatePartition;
    var o : OutputBitPartition) : StatePartition;

  calculates the state partition containing the information that should be present in the state partition and the returned state partition in a cluster. In the equation

  \[ \pi \cdot \tau_{O-B} = M_{S-O-B}(\pi) \cdot M_{S-S}(\pi) \]

the partition \( M \), that is the multiplication of the \( M_{S-S} \) and \( M_{S-O-B} \) partitions is calculated. This is done directly from the machine definition and using the equation

\[ \text{set}(M) \text{ if for all } i \text{ delta}(i,s) = \text{delta}(i,t)(\pi) \]

and for all \( k \) active in \( \pi_{O-B} \) \( \lambda_k(s) = \lambda_k(t) \)

to be able to calculate the result with a machine containing don’t care information in the next state table or the output table.

Input: NextStateTable and OutTable for the machine definition.

\( \pi \): The state partition.

\( o \): The output partition.

Output: CalcNSOB_SSpart: The resulting state partition.

**********************************************************************************

procedure clearcaBt (var cast : FullyConnectedSet);

Returns the dynamic memory used by cast.

**********************************************************************************

procedure findfullyconnectedsubsets (TauPtr : StateDemandPtr;
    var cast : FullyConnectedSet);

The modulus of a state demand partition is determined using a graph description of the demands. A node in the graph is a state. A connection between two nodes means that the states represented in both nodes should not be in the same block of the partition. The modulus is the number of nodes in the largest fully connected subgraph, a subgraph graph in which all nodes are connected with all other nodes. In this procedure all fully connected subgraphs formed by the state demands in list TauPtr are calculated. The nodes in these fully connected subgraphs are stored in cast.

Input:

\( \text{TauPtr} \): A list of pairs of states \( s \) and \( t \) which should not be in the same block of tau.

Output:

\( \text{cast} \): cast contains all nodes of a fully connected subset.

**********************************************************************************

function fullconnect (s : StateSeries;
    inset : statemset;
    Tptr : StateDemandPtr) : boolean;

checks whether \( s \) is fully connected with the states in set inser

end: (fullconnect )

begin

[ construct fully connected subsets, store them in cast ]

end: (findfullyconnectedsubsets )

**********************************************************************************

function StateDemandModulus (cast : FullyConnectedSets : StateSeries:

Calculates the minimal modulus of a partition \( \tau_{O-B} \) which is determined by the demands in list \( T \). The modulus is determined using a graph description of the demands. A node in the graph is a state. A connection between two nodes means that the states represented in both nodes should not be in the same block of the \( \tau_{O-B} \)-partition. The modulus is the number of nodes in the largest fully connected subgraph stored in cast.

Input:

\( \text{TauPtr} \): A list of pairs of states \( s \) and \( t \) which should not be in the same block of tau.

Output: StateDemandModulus: the returned modulus.
function ModSumStateDemands (cset1, cset2 : FullyConnectedSet) : StateSeries;

Calculates the minimal modulus of the sum of the two state demand partitions defined by T1 and T2. To do this the fully connected subsets of T1 and T2 are calculated and stored in cset1 resp. cset2. The required modulus is the maximum number of nodes in the intersection of a subset of cset1 with a subset of cset2.

Input: cset1, cset2 : The state demands.
Output: ModSumStateDemands : The modulus of the summed state partitions defined by the state demands in cset1 and cset2.

******************************************************************************

function ModMultStateDemands (cset1, cset2 : FullyConnectedSet) : StateSeries;

Calculates the minimal modulus of the multiplication of the two state demand partitions defined by T1 and T2. To do this the fully connected subsets of T1 and T2 were calculated and stored in cset1 resp. cset2. The required modulus is the maximum number of nodes in the union of a subset of cset1 with a subset of cset2.

Input: cset1, cset2 : The state demands.
Output: ModMultStateDemands : The modulus of the multiplied state partitions defined by the state demands in cset1 and cset2.

******************************************************************************

procedure CalcStateDemand (var StatePart, var Outpart, var TDem, var cset, var tmod : StatePartition; var OutputBitPartition; var StateDemandPtr; var FullyConnectedSet; var StateSeries);

Demands for the partition tau are calculated using:

StatePart * tau <= M_S-S (StatePart) * MSOB (Outpart).

Further the modulus of tau is returned in tmod.

Input : StatePart, y, the global variable MOutput and the machine parameters in NextStable and UTable.
Output : TDem, with the list of demands and tmod, with the minimal modulus.

begin
    // initialise TDem
    if TDem=nil then
        ClearStateDemand (TDem);

    // calculate the multiplication of the M-partitions
    MState := CalcSON_SSpart (StatePart, Outpart);

    // calculate the demands for tau and store them in TDem :
    if s<e (tau) if and only if s<e (StatePart) and s<e (MState) then
        (calculate the set of fully connected subsets of TDem )
        findfullyconnectedsets (TDem, cset);

        // calculate the modulus of TDem
        tmod := StateDemandModulus (cset)
    end; // CalcStateDemand
}
module test;
procedure SearchName (var NameT: Nametable; 
nm : integer; 
var name : wrkString; 
var len : integer); 
{ the number num is searched in NameT and the corresponding 
name is returned }
end;
procedure wrname (var f : text; 
var name : wrkString; 
len : integer); 
{ writes name name with length len to file f }
end;
procedure wrSpaT (var f : text; 
var Spat : Statepartition); 
{ writes partition Spat to file f. The number on position a gives the block 
number of state a in Spat }
end;
procedure wrOBitpart (var f : text; 
var p : OutputBitPartition); 
{ writes the output bit partition p to file f. If the number on position y 
is 0 bit y is in the don’t care block, if it is 1, bit y is important. }
end;
procedure wrIBItpart (var f : text; 
var p : InputBitPartition); 
{ writes the input bit partition p to file f. If the number on position x 
is 0 bit x is in the don’t care block, if it is 1, bit x is important. }
end;
procedure TestClusters (var f : text; 
wrcluster : ClusterPtr); 
{ for each cluster in the list wrcluster the input bit partition, output bit 
partition, state partition, modulus tau and tau demands are written to file f. }
end;
procedure TestManodepartitions (var f : text); 
{ the state partitions MOutput [j] and MInput [k] are written to file f. }
end;
procedure TestDefTable; 
{ The file with the name 'testdefTable' is opened. 
The matrices InputMatrixDef, OutputMatrixDef, containing the bit patterns are 
written to this file. 
Also all state numbers with corresponding state names are written. 
And last but not least the OutTable and NextStTable are written to this file. }
end;
Eindhoven University of Technology Research Reports

Faculty of Electrical Engineering

PREDESIGN OF AN EXPERIMENTAL (5-10 MW) DISK MHD FACILITY AND PROSPECTS OF COMMERCIAL (1000
MW) MHD/STEAM SYSTEMS.

(244) Kloonstra, Martin and Ton van den Boom, Ad Damen
A COMPARISON OF CLASSICAL AND MODERN CONTROLLER DESIGN: A case study.
EUT Report 90-E-244. 1990. ISBN 90-6144-244-3

(245) Berg, P.H.G. van de
ON THE ACCURACY OF RADIOWAVE PROPAGATION MEASUREMENTS: Olympus propagation experiment.

(246) Maagt, P.J.I. de
A SYNTHESIS METHOD FOR COMBINED OPTIMIZATION OF MULTIPLE ANTENNA PARAMETERS AND ANTENNA
PATTERN STRUCTURE.

(247) Jazwinski, L. and T. Spasova-Kassital
DECOMPOSITIONAL STATE ASSIGNMENT WITH REUSE OF STANDARD DESIGNS: Using counters as sub­
machines and using the method of maximal adjacencies to select the state chains and the
state codes.

(248) Hooijmakers, M.J. and J.M. Vleeshouwers
DERIVATION AND VERIFICATION OF A MODEL OF THE SYNCHRONOUS MACHINE WITH RECTIFIER WITH TWO
DAMPER WINDINGS ON THE DIRECT AXIS.

(249) Zhu, Y.C. and A.C.P.M. Reekx, P. Dykhoff
MULTIVARIABLE PROCESS IDENTIFICATION FOR ROBUST CONTROL.

(250) Pfaffenböger, P.M. and P.J.M. Cluitmans, H.M. Kuipers
EMODIS: Design and formal specification of a datamodel for a clinical research database
system.

THE ASCIS DATA FLOW GRAPH: Semantics and textual format.

(252) Chen, J. and P.J.I. de Maagt, M.H.A. Herbea
WIDE-ANGLE RADIATION PATTERN CALCULATION OF PARABOLOIDAL REFLECTOR ANTENNAS: A comparative
study.

(253) Haen, S.W.H. de
A PWM CURRENT-SOURCE INVERTER FOR INTERCONNECTION BETWEEN A PHOTOVOLTAIC ARRAY AND THE
UTILITY LINE.

(254) Velde, M. van de and P.J.M. Cluitmans
EKG ANALYSIS FOR MONITORING OF ANESTHETIC DEPTH.
(255) Smolders, A.B.
AN EFFICIENT METHOD FOR ANALYZING MICROSTRIP ANTENNAS WITH A DIELECTRIC COVER USING A SPECTRAL DOMAIN MOMENT METHOD.

(256) Hackx, A.C.P.M. and A.A.H. Damen
IDENTIFICATION FOR THE CONTROL OF HMDO INDUSTRIAL PROCESSES.

(257) Maag, P.J.I. de and H.S. ter Vorsche, J.L.M. van den Broek
A SPATIAL RECONSTRUCTION TECHNIQUE APPLICABLE TO MICROWAVE RADIOMETRY.

MANUFACTURING AND CHARACTERIZATION OF GaAs/AlGaAS MULTIPLE QUANTUMWELL RING WAVEGUIDE LASERS.

(259) Orlov, V.B.
DEFECT MOTION AS THE ORIGIN OF THE I/F CONDUCTANCE NOISE IN SOLIDS.

(260) Hoosijargs, J.R.
ALGORITHMS FOR SPEECH CODING SYSTEMS BASED ON LINEAR PREDICTION.

(261) Hackx, A.C.P.M. and A.A.H. Damen, Martin Kloostra
IDENTIFICATION FOR ROBUST CONTROL USING AN $H_{\infty}$ NORM.

MANUFACTURING AND CHARACTERIZATION OF GaAs/AlGaAs MULTIPLE QUANTUMWELL RING WAVEGUIDE LASERS.

(263) Freriks, L.W. and F.J.M. Cluitmans, M.J. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-) behaviour in relation with brainstem auditory evoked potential patterns.

MANUFACTURING AND CHARACTERIZATION OF GaAs/AlGaAs MULTIPLE QUANTUMWELL RING WAVEGUIDE LASERS.

(265) Hackx, A.C.P.M. and A.A.H. Damen
IDENTIFICATION FOR THE CONTROL OF HMDO INDUSTRIAL PROCESSES.

(266) Joxwiek, L. and A.P.H. van Dijk
A METHOD FOR GENERAL SIMULTANEOUS FULL DECOMPOSITION OF SEQUENTIAL MACHINES: Algorithms and implementation.