FSM information flow analysis for general decomposition with dispersed generation

Ensinck, J.C.L.

Award date:
2007

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 886

Master's Thesis

FSM INFORMATION FLOW ANALYSIS FOR GENERAL DECOMPOSITION.

Hans Ensinck

Supervisors:	Prof.dr.ir. R.H.J.M. Otten
	Dr.ir. L. Józwiak
Coach:	D. Gawlowski, M.Sc.
Date:	May 2007

Abstract

The rapidly increasing complexity of digital systems and growing quality demands related to them result in an increasing need for high-quality EDA tools for synthesis of these systems. Specifically, the synthesis of the application specific controllers requires special attention, because the controllers cannot be synthesized from standard high-level components, but must each time be synthesized anew from basic logic elements.

Fortunately, the modern platform FPGAs allow for fast and cheap development and implementation of a growing share of the modern complex digital systems. However, their benefits cannot be fully exploited by the traditional synthesis methods and tools. Therefore, new better synthesis methods and tools are needed, specifically for the controller (FSM) synthesis.

The subject of this report is a new method for an effective and efficient controller synthesis, based on the information-driven approach to digital circuit synthesis, the theory of general decomposition and the theory of information relationships and measures. This method consists of the following three main phases: FSM information flow analysis, molecular FSM clustering and FSM network construction. This report discusses the first two phases, when focussing on information flow analysis.

The method is implemented in a prototype synthesis tool SeMaDe. After analysis of the initial implementation of the method, it turned out that several of its algorithms should be improved or replaced by new better algorithms. The main aim of this master project was to develop several new or modified better algorithms for the FSM information flow structure analysis and their actual software implementation in C++. This included testing of the information flow analysis algorithms and their software implementation.

In the first phase, the FSM information flows of the machine are analyzed and the input, state and output information is structured into portions called molecules that are efficient regarding both the production and consumption of information. Some of the created portions are still too small too become efficient partial machines or are similar to each other. Therefore, the clustering phase is needed to combine the molecules that are too small and / or very similar, resulting in a smaller number of macro molecules that will eventually become the actual partial machines. The third and final phase, the network construction, will then create the actual network of partial machines, which is the goal of the entire method.

To test the first two parts of the method and the EDA tool implementing the method, we ran several hundreds of benchmarks and analyzed the produced decompositions. The extensive testing confirms that the tool works correctly and produces valid decomposition structures for all the analyzed cases. For all the test cases that we were able to analyze regarding the result quality, the tool produced very good results.

However, at this stage, the final benchmarking and comparative research, cannot yet be performed, because the final part of the whole method is not yet implemented as it should be which makes it impossible to automate the testing and result analysis. Once the final phase, the construction of a network of partial machines, is fully implemented, the tool will be able to produce the actual FSM network specifications in a format understandable to other tools. Once this is achieved, benchmarking can be automated and the method and tool can be compared to other methods and tools on many benchmarks. This will finally confirm if the method is successful. However, this was not the subject of this master project. The aims of this master project are fully achieved.

Acknowledgements

I would like to thank the people from the TU/e ICS/ES department for giving me the opportunity of making this thesis. Specifically, I would like to thank Dr. Ir. L. Jóźwiak, my supervisor and M.Sc. Dominik Gawlowski for our collaboration on this project. Also I would like to thank Prof. Dr. Ir. R. Otten, Dr. Ir. T. Basten and Dr. Ir. L. Jóźwiak for being in my graduation committee. Consequently, I would to thank the secretaries of the ES group for helping me out with some organisational issues and my fellow students for the many nice conversations we had during my master project. Finally, I wish to express my thanks to my family and friends, who have supported me during my time at TU/e and my graduation.

Eindhoven, May 2007

Hans Ensinck

Contents

List of figures

1. Introduction

The first chapter of this report contains an introduction to the subject of the master project reported. In paragraph 1.1 the context of this work is described. It is explained why this project has been performed and where its results can be applied. In the second paragraph 1.2, the problem that is to be solved is stated. A short overview of the complete solution method is given in section 1.3. The final section of the introduction describes the aims and the subjects of my master project and of this master thesis. In the main lines, my master project assignment consisted of a participation in the development of the last version of the algorithms for the first two parts of the method and the actual software implementation in C++ of the FSM information flow analysis algorithms.

1.1 Context

Application specific embedded systems are used in virtually all fields of human activity and play a remarkable role in today's life. The share of embedded systems with low or medium production volumes continues to grow. The fast growing complexity and the critical character of systems in many embedded applications impose extremely high quality requirements. Some of the most important goals of industry are the reduction of costs and time to market, while simultaneously increasing the power, complexity and quality of systems. It is therefore very important to appropriately organize and automate the embedded system design process.

In this design process the following four main abstraction levels can be distinguished:

1. The system specification level: At this level the entire system is defined. This is done by describing the interfaces to the "external world", the system's function and parametric requirements (e.g. timing, cost).
2. The architectural level: At this level is decided how the system required is build up of a number of sub-designs that each have a particular function in the complete design, that communicate with each other. Collaboration together, the sub-systems realize the required behavior and satisfy the parametric requirements
3. The logical level: The logical level defines how the sub functions should be implemented as a network of logic-level elements (e.g. gates, LUTs, FFs) and their interconnections.
4. The physical level: Defines how the design is actually implemented on a chip with the technology primitives.

The subject of this master project is related to the lowest-level architecture synthesis and logic synthesis. The most important design task at the architectural level is to find an appropriate structure, composed of sub-systems (architecture modules), which (after its implementation) realizes the specified behavior of a system, satisfies given (physical, economical etc.) constraints and optimizes certain objectives. The most important task of logic synthesis is the translation of the symbolic, functional description of an architecture module into the binary logic description of a network of logic components that realizes the behavior of the module. This network must satisfy specified constraints and optimize given objectives. In other words, the architecture synthesis decomposes a system in a (near-) optimal structure of cooperating architecture modules, and the logic synthesis decomposes further each architecture module in a (near-) optimal network of logic building blocks.

1. Introduction

In recent years, there is a fast growing usage of modern field programmable devices like FPGAs and CPLDs. The reason for this is that they have numerous advantages and are especially well suited for implementation of application-specific embedded systems that are produced in small or medium-large series such as embedded controllers in medical, test and measurement instrumentation, machines, etc. or applications that require re-programmability.

Unfortunately, the traditional architecture and logic synthesis methods and tools are not adequate for these types of devices, because they considerably differ from the traditional synthesis targets. This is because these new devices have limited number of inputs and outputs, limited internal memory and limited communication channels, etc. in contrast to the more traditional devices where functional complexity is the main constraint. These traditional synthesis methods cannot cope well with these hard constraints and only explore a small part of the space of possible solutions.

Another problem of these traditional methods is that the final target is not taken into account while doing the initial synthesis, and so a post synthesis net-list partitioning and a technology mapping is required. However, if the actual programmable logic target (e.g. FPGA) strongly differs from the simplified initial synthesis proxy target, the technology mapping cannot guarantee good results, because this initial synthesis is performed without close relation to the actual target. This all results in inferior designs and often can also result in serious problems with the satisfaction of some physical constraints.

A new approach to solve this problem that is much more promising is the information-driven approach based on general decomposition and information relationship measures that has been proposed by Dr. Ir. L. Jóźwiak. In the proposed information-driven approach to circuit synthesis, the distribution, processing and transmission of information play a central role. The circuit is constructed in such a way that information flows in the circuit are ordered according to the information production and consumption, appropriately combined, compressed and kept as local as possible. Hard constraints imposed by the limited number of inputs and of the logic building blocks (LUTs and CLBs) are satisfied by explicitly constructing the sub-functions that directly fit in the logic building blocks.

As a result, particular sub-functions have small number of inputs and outputs, the sub-networks for particular outputs converge rapidly and satisfy hard constraints of logic blocks, interconnections are minimized, and the resulting circuit is fast and compact.

The approach relies on analysis of the information structure and information flows in the sequential machine or a combinational function to be implemented, as well as in the circuit under construction, and usage of the results of this analysis to control the synthesis of a circuit that implements the function. To enable qualitative and quantitative analysis of the information structure and information flows, an adequate analysis apparatus is necessary, which facilitates the following:

1. Analysis of the information flows – where and how a particular information is produced, and where and how it is consumed,
2. Analysis of the relationships (similarity, difference) between various information flows,
3. Introduction of the quantitative flavor (quantity, importance weight) to characterize the analyzed information flows and their relationships.

All these requirements are fulfilled by the apparatus of information relationships and measures.

The decomposition paradigm is successfully used in many fields of engineering and is a way of simplifying a complex problem by breaking it down into a number of sub-problems of lower complexity. In the case of the circuit for Finite State Machines (FSMs) this means to find a network consisting of a number of component machines that together implement the output behavior of the originally specified FSM.

Using this information-driven approach based on the general decomposition theory [6], [17], [22] and the apparatus of Information Relationships and Measures [7] as proposed by Dr. Ir. L. Jóźwiak already some very good results were achieved in the form methods and prototype tools for similar problems:

- SeCoDe by L. Jóźwiak and A. Slusarczyk for state encoding of FSMs [17], [22].
- IRMA2FPGAS by L. Jóźwiak and A. Chojnacki for synthesis of combinational circuits [1], [2].
- SeMaDe by L. Jóźwiak and P. Konieczny as the first method and preliminary prototype tool for general decomposition of FSMs [19].

These results stimulated the research group of Dr. Jóźwiak and STW to start a project (EES.5766) related to an effective and efficient architecture and logic synthesis of application-specific embedded controllers for programmable hardware implementation. In this research several persons are directly involved to different degrees: Dr. L Jóźwiak (project leader), D. Gawlowski (Ph.D. project), myself (ing. H. Ensinck for my master project), as well as A. Slusarczyk and A. Chojnacki (former Ph.D. students of Dr. Jóźwiak).

My particular role in the project consisted of a participation in the development of the last version of the algorithms for the first two parts of the related FSM decomposition method, and the actual software implementation in C++ of the FSM information flow analysis algorithms. My role will be precisely presented further in this report.

1.2 Problem description

Almost all (complex) modules of digital designs have a processing path (or data path) / controller structure. Here a processing / data path processes input data to output data and the controller monitors the status signals and produces the control signals from and to the data path. The structure as described above is referred to as the Glushkov model (see Figure 1.1). An entire design can consist of just one such structure. But these structures can also be put in series or parallel or they can be nested in each other. For instance, a unit from the data path could have a number of data path units and some controllers of its own. The data path of a digital design consists of a number of (basic) units / building blocks (like adders, shifters, registers, multipliers, etc. or on a higher level complete algorithms like FFT's or different types of de/encoders, etc.).

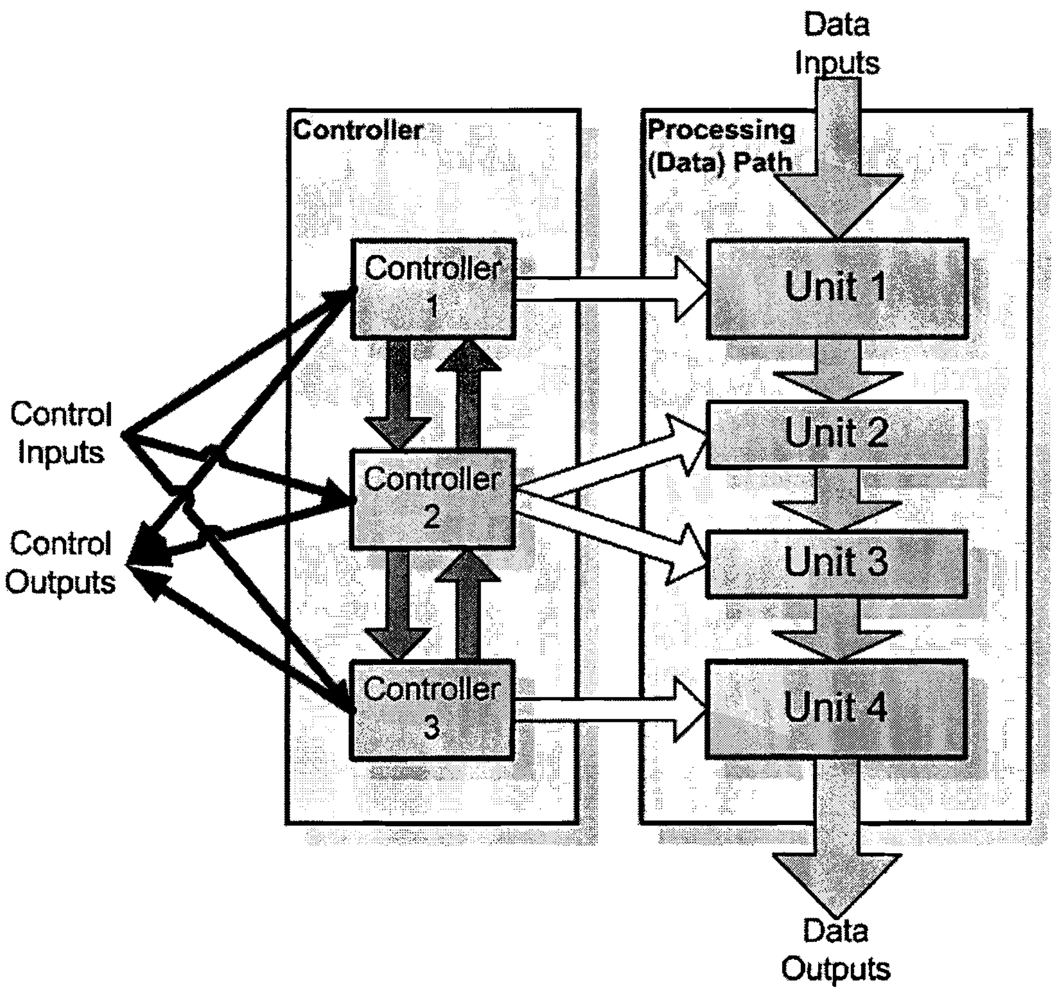


Figure 1.1: An illustration of the Glushkov model (with an example of a possible internal structure).

Because these units/building blocks of the data path generally perform standard functions, they can usually be reused. Therefore, data-path block synthesis for a specific operation has (in principle) to be performed only once and therefore a lot of effort can be put into finding a good implementation of the data-path building blocks. They could e.g. be put into a (post-synthesis) library, so when needed in a design already synthesized versions of the building blocks can be used directly. This does not mean that some units of the data path could not be application-specific and therefore especially designed and synthesized for the specific application.

A controller on the other hand, is always application-specific because in organized and controls the work of the data-path for just a particular application. Consequently, a controller has to be synthesized for each different application anew (and every time a controller is changed). Therefore an automatic synthesis tool is here extremely important that produces very good synthesis results in terms of circuit speed, area and power consumption using only a reasonable amount of computation time and memory space.

As mentioned before (in the previous section) the problem of synthesizing an embedded application-specific controller can be solved by using the information-driven decomposition paradigm where a sequential machine is split up into a number of component machines that together realize the input-output behavior of the original machine. At the same time the given constraints must be met and the given objectives must be optimized. The theory of general decomposition states the necessary and sufficient conditions (and gives the necessary constraints) for an abstract network of component machines to be a valid realization of a given sequential machine. In this way is defines a generator of all (possible) correct circuit structures. This alone however does not solve the practical decomposition problem. For larger machines for which so many (valid) decompositions exist that an exhaustive search is not possible (the solution space is too large). Therefore, heuristic search algorithms must be used to construct only the most promising decompositions in such a way that a (near)optimal solution is found, while drastically reducing the computation time and memory usage otherwise taken by an exhaustive search.

Because of the large freedom that is left (because there are very many possible decompositions), the synthesis process has to be steered toward an (near) optimal solution. For this purpose, in the first place, the relevant information about the original machine must be used.

Using for instance the apparatus of Information Relationships and Measures, the internal information flows of the machine can be examined. In this way, it is possible to see where and how certain information is produced and / or consumed, and how difficult it is to produce some portion of information. With this analysis, coherent pieces of information are found that should be produced or consumed together. By putting one or more of these coherent pieces of information together in a component machine, a decomposition is found which exploits the internal structure of the original machine well, and will therefore result in a nice decomposition of relatively simple component machines with only few interconnections between them.

Besides the internal information, also information on the external relationships of the sequential machine should be used to steer the decomposition towards a (near) optimal solution. By external information is meant e.g. information on where (physically) the particular FSM inputs and outputs are connected, or from/to what part of the data path certain signals are received/transmitted (see figure 1.1). By taking this information into account, a different decomposition could be found then when only considering the internal structure of the controller (e.g. it could be better to produce / consume certain information near a part of the data path where many of these signals come from, or near some input / output pins. It could also be better to produce some information more than once if the production is cheaper than the wiring otherwise required to transmit this information to its destination).

Using these two sorts of information (i.e. on the internal as well as the external information flows) and confronting them with the actual synthesis objectives and constraints, heuristic decisions will be taken to control the correct-by-construction circuit generator provided by the general decomposition theorem, to only synthesize the most promising circuit structure(s).

Summing up, the aim of the while research is to develop a method that will be worked out to precise algorithms implemented as a C++ program of a CAD tool (SeMaDe) to find one or more (near-)optimal decompositions of a machine. This tool however, will only be a part of a larger tool chain that will produce a synthesized version of a given controller description. An example of such a tool chain is given in figure 1.2.

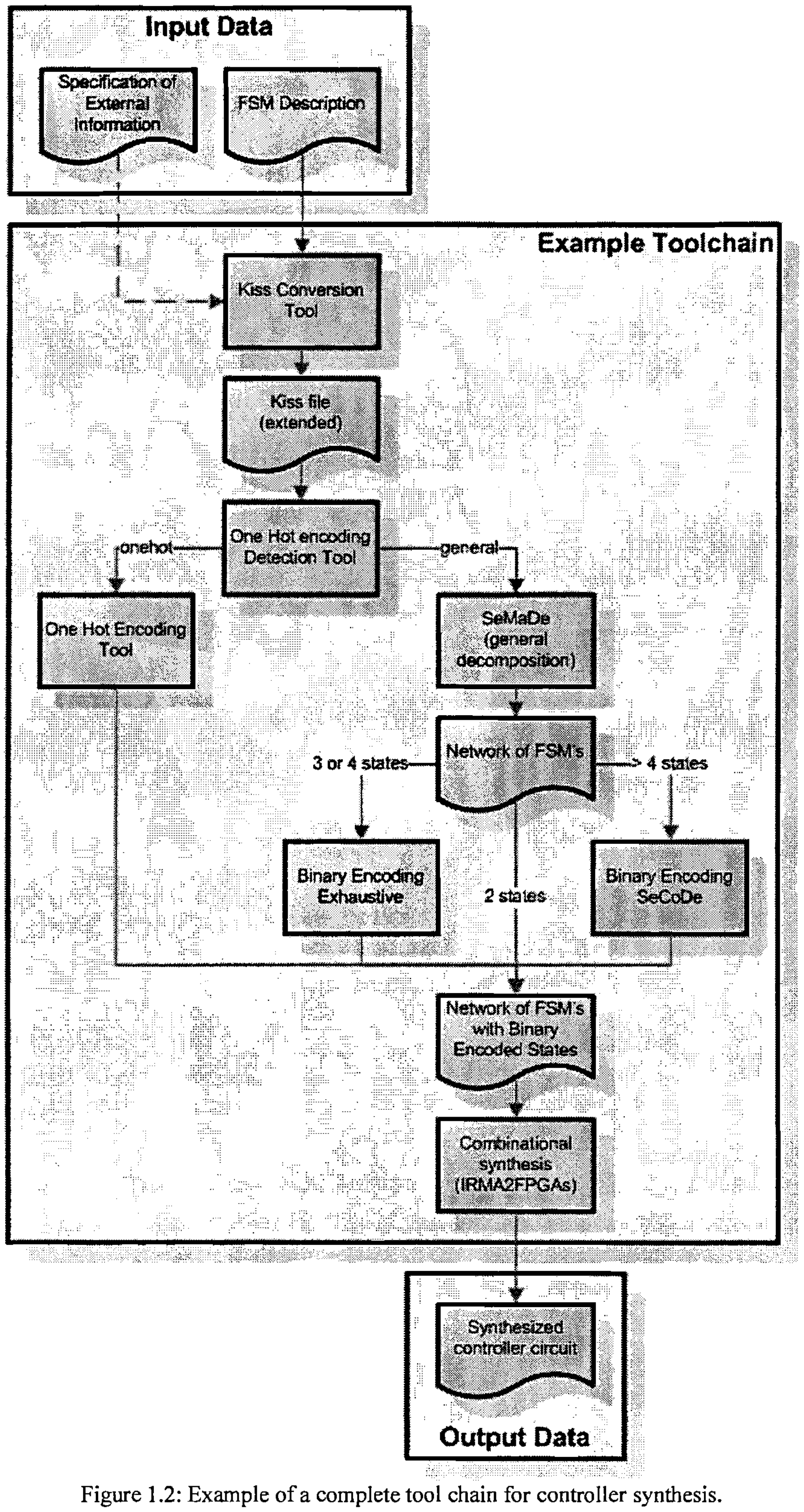

Input Data
Specification of External Information
FSM Description
Example Toolchain
Kiss Conversion Tool
Kiss file (extended)
One Hot encoding Detection Tool
onehot
general
One Hot Encoding Tool
SeMaDe (general decomposition)
Network of FSM's
3 or 4 states
> 4 states
Binary Encoding Exhaustive
2 states
Binary Encoding SeCoDe
Network of FSM's with Binary Encoded States
Combinational synthesis (IRMA2FPGAs)
Synthesized controller circuit
Output Data

Figure 1.2: Example of a complete tool chain for controller synthesis.

1. Introduction

The input to this tool chain is a functional description of a sequential machine. This can be of any form such as a hardware description language like VHDL or Verilog. It can also be in the form of a graph of states and transitions or an RTL level description. Further, an optional specification of the external information can be given (as explained earlier). In this file constraints can be defined in terms of delay, set- and hold-times, etc for certain signals or groups of signals. However if this information is not available the controller will be decomposed using only information on its internal information flow structure.

A "Kiss conversion tool" will convert the specific input file(s) to an (extended) Kiss file format as specified by UC Berkeley. A standard Kiss file can be used to define any sequential logic circuit with binary encoded inputs and outputs. Even when it has non-deterministic behavior (don't cares in the next state function) or not completely defined outputs (don't cares in the output function). If this tool is provided with external information about the controller, the Kiss file will be extended such that this one file contains all information required for the tool chain to synthesize the controller.

Once this file is created, it is determined what type of binary encoding for the states is optimal. Research performed by the section of Dr. Jóźwiak with thousands of benchmarks has shown that optimal state encoding is with close to (near-)minimum number of the binary encoding variables or a (near) one hot encoding in a large majority of cases.

A minimum-length encoding means that the number of state variables / two state memory elements (in case of binary logic) is $\lceil \log_2 |S| \rceil$, where $|S|$ is the number of states of the particular machine (cardinality of the state set). This number of state variables is the minimal number of bits needed to uniquely to encode all states. This encoding will result in the minimal number of memory elements and binary next-state functions and thus the next-state and output logic with a minimized number of inputs, but with probably higher functional complexity.

Secondly, the sections demonstrated that for one type of machines, namely, the state dominated controllers with very simple sequential behavior (not many branches from a particular state) like purely sequential controllers, (close to) one-hot encoding could yield very good results. In one hot encoding (almost) every state is assigned to its own state variable. So this encoding actually will result in the maximal number of memory elements. However because of the purely sequential behavior of this type of controller the number of inputs to the particular next-state and output logic functions is very small and also their functional complexity is extremely low. This type of encoding is extremely simple: assign a state variable to each state symbol. Based on the research results of the section of Dr. Jóźwiak, it is easy to decide in which cases one-hot encoding should be applied.

If, on the other hand, is found that the given controller description can be encoded very well with a binary encoding. The tool SeMaDe can be called. This is the tool described in this report. It will create a general decomposition of an original FSM in the form of a network of small finite state machines. However these FSMs do not require all to have only two states (which would mean a complete binary encoding). This means that the general decomposition method and tool described in this report creates a multi-valued (partial binary) encoding representing architecture of the original FSM.

Some machines however could only have two states, so for these FSMs encoding is directly established because there are only two ways to encode these two states (called A and B): A=0, B=1 or A=1, B=0. These encodings are each others inverse, and because inversion in an FPGA is a free operation there is only one (fundamentally different) encoding.

For decomposed machines of 3 or 4 states, (at least) two state variables of two blocks are needed for the encoding. An exhaustive search is therefore possible, because there are only a few ways to encode them. For a FSM with 3 states (A, B and C) these are the six possible encodings.

In the case of a binary encoding for 3 states, this is defined by a product of two two-block set systems of which the product results in $\left\{\overline{A};\overline{B};\overline{C}\right\}$ (for more information on set systems and the product operation on set system the reader is referred to chapter 2). The six possible encodings are as follows:
$\left\{\overline{AB};\overline{C}\right\}\cdot\left\{\overline{AC};\overline{B}\right\}$, $\left\{\overline{AB};\overline{C}\right\}\cdot\left\{\overline{A};\overline{BC}\right\}$, $\left\{\overline{AB};\overline{C}\right\}\cdot\left\{\overline{AC};\overline{BC}\right\}$, $\left\{\overline{AC};\overline{B}\right\}\cdot\left\{\overline{A};\overline{BC}\right\}$,
$\left\{\overline{AC};\overline{B}\right\}\cdot\left\{\overline{AB};\overline{BC}\right\}$ and $\left\{\overline{A};\overline{BC}\right\}\cdot\left\{\overline{AB};\overline{AC}\right\}$.

In the case of a binary encoding for 4 states (A, B, C and D), an encoding is also defined by a product of two two-block set systems of which the product results in $\left\{\overline{A};\overline{B};\overline{C};\overline{D}\right\}$. In this case however there are only three possible encodings because each block can only contain one state (otherwise a product of two two-block set systems could never result in a set system which uniquely defines all four states). The three possible encodings are as follows:
$\left\{\overline{AB};\overline{CD}\right\}\cdot\left\{\overline{AC};\overline{BD}\right\}$, $\left\{\overline{AB};\overline{CD}\right\}\cdot\left\{\overline{AD};\overline{BC}\right\}$ and $\left\{\overline{AC};\overline{BD}\right\}\cdot\left\{\overline{AD};\overline{BC}\right\}$.

Machines with more than four states will require at least 3 state variables and the number of possible encodings will explode exponentially. Therefore, we should use the (already developed) SeCoDe (or any other binary encoding tool), although SeCoDe can also be used for the partial machines with only 3 or 4 states.

SeCoDe (like SeMaDe) is also based on the theory of general decompositions and information relationships and measures but it analyses the structure of a machine at the finest level, by determining which input and atomic / elementary state information are required to compute a particular output or an atomic / elementary state information while SeMaDe can also work with larger portions of information. The second difference is that while SeMaDe always searches for a natural multi-valued decomposition to synthesize the actual circuits that will implement the original FSM, SeCoDe searches directly for the best binary state encoding so the output of SeCoDe is guaranteed to be a network of only two state partial machines. So while SeCode will not always find a (close to) the best natural decomposition and will require more computation time, for the partial machines generated by SeMaDe it is an exceptionally good tool since it will structure the machine further where SeMaDe could not decompose further into smaller machines. The computation time is always reasonable, because the partial machines generated by SeMaDe are always of a small size.

The result of this final encoding step is again a network of sequential machines, however now all machines have two states and so the binary encoding of the states is complete. Because we assumed that the inputs and outputs where already binary encoded, the next state and output functions of all machines, as well as their input encoders and output decoders are defined as logic functions with binary input and output bits, so any combinational logic synthesis tool can be called. In Figure 1.2, the IRMA2FPGA (Information Relations and Measures Applied to FPGA) tool is mentioned for this purpose. This tool is also based on the theory of general decomposition and information relationships and measures and yields very good results. However any other commercial or academic logic synthesis tool can be used for this purpose.

Now we have a network of sub-functions and memory elements which can be directly mapped onto the target architecture. Of coarse, placing and routing still has to be performed using the tools from the vendor of the target device.

1.3 Solution concept

To solve the problem of the FSM architecture synthesis described in the previous section, we will use the information-driven general decomposition approach to digital circuit synthesis. This is the same approach as applied in some other methods and prototype tools (e.g. Devisa and SeCoDe [17], [22] and IRMA2FPGAs [1], [2]) developed under leadership of Dr. L. Jóźwiak for solving similar problems. The architecture synthesis and network construction method to solve the problem described in the previous section has been already earlier proposed and implemented in the first prototype version of the SeMaDe tool [19]. It consists of the following three main phases:

1. The information flow analysis phase,
2. The clustering phase, and
3. The final FSM network construction phase, where the clusters (found in the second phase) are converted into a network of adequately interconnected component machines.

The first prototype version of SeMaDe has been developed to mainly check all the concepts and although it is a compete tool that implements all the analysis, clustering and network construction steps and produces valid FSM networks, the implementation of particular parts of the tool was preliminary and required further research and development effort.

Trough thoroughly analyzing the first version of the SeMaDe, Dr. Jóźwiak discovered several inefficiencies in some of its algorithms and proposed related improvements. It was also necessary to implement it consistent with SeMaDe tool for the (final) binary FSM encoding. This led to formulation of the PH.D project in which SeCoDe was developed and the STW research project of which the work described in this report is a part. Further development of the information flow analysis part of the SeMaDe tool was the main aim of the master project being the subject of this report.

In consequence of further works, several algorithms in the first two phases of the last version of SeMaDe being the subject of this report, at least partially differ from those in the earlier developed methods (as mentioned above) for solving similar problems. In result, the FSM architecture synthesis method as implemented in the first prototype of SeMaDe remained unchanged at the method level, but several algorithms of its particular parts and their corresponding software implementations are substantially different from those implemented in the first SeMaDe prototype and other tools as mentioned above for similar purposes.

The main general aim of the decomposition process is to distribute the production of the state information amongst the particular partial machines in such a way that it is well structured from the following two viewpoints:

a. ***State Information Production:*** The state information should be structured in portions that are convenient for the production of each particular portion, i.e. in such a way that:
   i. the amount of the input and imported state information needed to produce the state information of each particular partial machine should be as low as possible, and

ii. the number and length of interconnections from the primary inputs and from other partial machines to a particular partial machine will be minimized.

b. ***State Information Consumption:*** the state information should be structured in portions that are convenient for the consumption of this particular state information by the primary outputs and other state information producing partial machines; this means among other things that the state information required by a particular output or another partial machine should not be distributed for its production among too many partial machines, but should be produced in as few as possible partial machines, if only this is not in contradiction with requirements of the convenient state information production.

The decomposition process aims to find the best match and tradeoff between the above two viewpoints and among the partial machines, to result in a high quality total solution.

In the process of designing digital circuits (or similar design processes) two different activities can be distinguished: analysis and synthesis. Analysis activities try to gather relevant information about the problem, to make it possible to make the right decisions for synthesis, which is deciding on how the design is actually implemented. Rather than seeing analysis and synthesis as two completely separate stages (where analysis is followed by synthesis), in our method we start with doing only analysis and gradually increase the amount of synthesis (thus reducing the amount of analysis) during the method (see figure 1.2). This is done by making synthesis decisions as soon as analysis provided us with enough information to justify this decision. The process of this decision-making as early as possible gives us extra information on which to base further decisions.

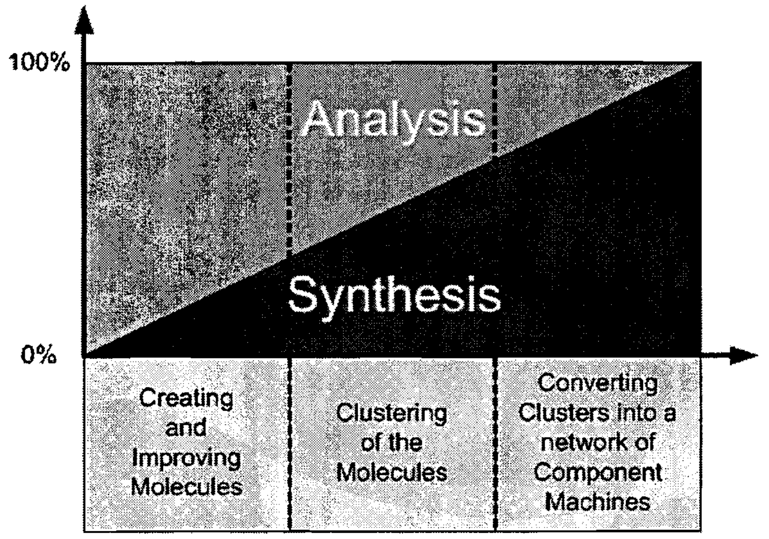

100%
0%
Analysis
Synthesis
Creating and Improving Molecules
Clustering of the Molecules
Converting Clusters into a network of Component Machines

Figure 1.3: The ratio of analysis and synthesis during the different phases of the method.

In the analysis phase, a sequential machine is first read (from a .kiss file). Thereafter, information molecules are created. These molecules (in analogy with material molecules in physics or chemistry) are some portions of information larger than atoms in the sense that a molecule contains more information than elementary or atomic information. Atomic information refers to a minimal amount of information which makes it possible, to distinguish two particular symbols [7]. The SeCoDe tool, for instance, works with the atomic state information (to perform the binary state encoding for FSMs), while SeMaDe directly creates molecules from the initial FSM specification and further works with these molecules.

Working with atoms has the advantage that you can precisely analyze what information is produced / consumed at a certain place. A disadvantage of working with atomic information is that it is more difficult and time consuming to make an adequate analysis of the information relationships. Also the following synthesis phases (e.g. clustering) will take longer because more and smaller components have to be merged.

Another disadvantage is that you tend to work with too fine portions of information. In this case the relationships between such small portions of information may not be as clear as the relationships between the larger molecules.

Working directly with molecules (as is done in SeMaDe) is faster but also less precise. The best option would be therefore, to base the method on molecules but to also do the analysis on much finer portions of information (e.g. atomic information) when needed. In this way we can profit from both approaches: A fast algorithm with a precise information analysis (and therefore good molecules).

1.3.1 Phase 1: Analysis

A molecule (as stated before) can contain more information than an atom, but it is still small compared to the whole FSM considered. As it will be more precisely (mathematically) described in the section of this report on the theoretical background, a molecule is defined by four pieces of information (input information, consumed state information, produced state information and output information). Molecules are used to model information flows in the FSM from the inputs and the current-state to the particular outputs, as well as, information processing within the molecules. They are used for both the information flow and processing modeling, as well as, in the role of the early prototypes of the actual partial machines in the decomposition, that corresponds to partial computations of the original FSM.

Two basic molecule types can be distinguished:

- Molecules A: created from the viewpoint of (an efficient) computation of particular portions of output information, as e.g. information for a particular binary output.
- Molecules B: created from the viewpoint of (an efficient) creation of particular portions of the state information.

The molecules come in four flavors:

- Mealy molecules A: These are related to computation of some output and possibly also of some state information.
- Split Molecules A: These are specific molecules A which are split according to their inputs. In this way not only the outputs but also the inputs are used to decompose the output information.
- Moore molecules A: These molecules produce information necessary for outputs which only require state information, so the output information needed for this particular output can be directly mapped onto its produced state information. Because of this fact, these molecules' main aim is to produce state information (which makes it similar to a B molecule), to produce a particular output (which makes it similar to an A molecule).
- And finally molecules B: These are related mainly on the efficient computation of state information (and possibly some output information). The outputs do not give information on how to encode the state information (just on how the output information computation

is originally structured), so the state information can only be decomposed well accounting for the inputs of the machine and the state-state dependences.

During the construction of molecules A, it is necessary to decide if the input information required by each of them will be delivered directly or after encoding of some inputs, and the corresponding input encoders have to be constructed.

The information flow analysis being the main subject of the work reported here is composed of the following main steps:

- reading the controller (FSM) specification from a KISS file;
- analysis of the input-output and state-output information flows and creation of molecules A;
- splitting the A molecules according to inputs if possible;
- creating the Moore molecules A for outputs which do not require input information;
- Finding a maximal SP set system to determine the state information to be computed;
- analysis of the input-state and state-state information flows and creation of B molecules;
- analysis and improvement of the created molecules;
- Selecting a limited set of the most promising subsets of molecules that each of them corresponds to a valid decomposition. Each subset represents a different alternative for clustering.

The complete FSM information flow analysis necessary to create adequate molecules should be composed of both the internal, as well as, the external information flow analysis. In the internal information flow analysis, the input to output, input to state, (present)state to (next)state and state to output relations are analyzed. Based on the results of this analysis, the molecules will be created.

In the method that we propose, the molecules A are created first, where each molecule A corresponds to one output bit of the original machine. This means that a molecule A has to compute enough (output) information to support a single output bit of the original machine. To do this, a particular molecule A requires information from (some of the) inputs (consumed input information) and some information about the state (consumed state information) of the original machine. With this information it is not only possible to compute some output information, but also some state information (produced state information) although this is not their main aim. The state information that is required by a certain molecule (consumed state information), but not computed by this molecule (produced state information), is called the imported state information.

After the output information is decomposed according to the outputs, it will also be tried to split the output information according to the inputs. It could happen that some (possibly elementary) portions of output information do not require all input bits / input information that is needed to compute the entire output. In this case each particular A molecule is decomposed to still produce all output information needed for the output bit but with all split A molecules each having the smallest possible input support. The output will then be computed with a virtual output decoder. The encoder is up to this point still virtual because it is not actually created, but it is suggested by the fact that these A molecules only compute part of the output information for a particular output bit. If however an A molecule contains at least one portion of atomic output information that requires the input information that was also needed for producing the entire output bit, splitting is not useful and the corresponding output will be computed by an un-split molecule A.

Then the Moore molecules are constructed for the outputs which do not require input information. This is a special case. The Moore molecules A do not need any output logic because the state information can be directly mapped on to the outputs. Secondly the produced state information (and also the output information) can be encoded on just one bit so it can be directly used for binary encoding of the states. This is also the main reason for their low cost. The aim for this type of molecules is to produce all state information needed for a particular output bit in one molecule (since the output computation for a Moore output in requires no input information). Of coarse, to do this, state information and input information is required. Secondly, these molecules can not be split using output decoders.

Now that the molecules A (complete and split) and the Moore molecules are created, the computation of all the output information for a valid decomposition is covered. Although some state information can already be computed by these type A molecules, possibly not all the state information for a valid decomposition can yet be computed. And even this is the case, it is anyway useful to analyze the computation of the state information from the viewpoint of the inputs. To determine what state information should (at least) be computed is not really obvious since this state information is fed back in sequential machines, because the next state is computed from the current state (and input information). So the maximal SP set system has to be determined, which is a the largest set system (contains the least amount of information) that has a substitution property. This means that the next state can be computed from the previous state thus making the decomposition valid regarding the state information and taking into account the recursive behavior.

Now that the maximal required SP set system is known, at least this state information has to be produced by molecules B. The molecules B where already mentioned several times before, and are essentially the same as molecules A, except for the fact that they (basically) do not produce output information. These molecules B will be constructed and also be improved so that they can maximally exploit the input information consumed.

By then, After the production and improvement of the A and B molecules the analysis phase is complete. From this point on, all molecules are treated equal (no difference is made between molecules A and B) and each molecule is treated as a cluster (containing just a single molecule).

The main subject of this report is the first part of the decomposition method briefly described up to this point and its software implementation. However this is just the first phase of the complete method to generate a good and valid decomposition of a sequential machine. For a better understanding of the complete decomposition method, the successive phases are also shortly addressed.

1.3.2 Phase 2: Clustering

After a wanted set of the alternative most promising initial subsets of molecules is created, the actual sub-machine network construction can start. Its first step being the second phase of the complete method, is clustering of some molecules in each subset. At the end of the clustering process, each macromolecule being its result, corresponds to a partial machine. In the beginning, all the molecule clusters contain only one molecule of a given subset. During the clustering, the clusters (which are not indicated by the analysis part to be untouched) are gradually merged based on a carefully constructed affinity measure corresponding to the actual implementation cost, in such a way that clusters with high affinity are clustered together. Although the concept of clustering is intuitive there are a lot of aspects to consider to get a good clustering (which eventually results in good component machines). The intermediate solutions generated which have the lowest implementation cost are considered for further clustering.

An affinity measure determines to what degree some clusters are similar and the weights of each of these measures are of high importance. If this is not done carefully clusters might be joined for the wrong reasons resulting in bad decompositions. The weighting of the different affinity measures is important because in this way a certain measure can be emphasized thus focusing on a certain property. Also the cost function which determines the quality of the solution has to be carefully defined. Finally the search parameters have to be adjusted well to steer the search in promising regions of the solutions space and keeping a low running time of the tool versus maintaining a high possibility that many very good solutions remain in the searched solution space. If clustering is stopped to early, there will be too many component machines and still al lot can be gained by further clustering because there are still some clusters with high affinities that are not merged. This can result in information that is produced more times than necessary. If on the other hand clustering is stopped to late clusters are merged that should not be clustered, this will also result in bad decompositions.

During the clustering, we have also to take care of the fact that sometimes the information necessary for computing a particular output bit can be present in two or more different clusters that should not be merged together, because of other reasons. In such a case, instead of trying to merge the otherwise not similar clusters, we can compute the partial output information portions, for this particular output bit in the separate clusters (partial machines), and then to combine this partial output information portions in the total output information of the particular output bit, using an output decoder. This way, similarly to the input decoders on the input side, the appropriate output decoders will be created on the output side.

Due to the heuristic character of this sort of problems and the uncertainty at this stage, analytical methods will not suffice. Also because of the possibly large size of this sort of problems, also exhaustive searches are no good alternative. So only considering one alternative at each stage is not very likely to lead to good final clustering solutions. Therefore this algorithm is implemented as a double beam. The principle will be further explained in Chapter 3. And its actual application to clustering is discussed in Chapter 4.

1.3.3 Phase 3: Network construction

When the clustering is finished, some final clusters are present in each alternative solution. Each final cluster is an abstract model of a component machine defined with the (next) state and output information that can be computed from some input and (present) state information. It must however still be converted into an actual corresponding component machine. To do this, it must be decided from where (i.e. which partial machines) each particular partial machine will import the missing information necessary for its computations and how it must be encoded (mapped onto a number of bits). This encoding is a tradeoff between the number of connections required between the component machines and the amount of computation necessary to get the required information encoded on these bits. Now the component machines can finally be interconnected. And as a result we have our network of component machines that together realize the output behavior of the original machine.

After the network construction phase, the decomposition will be complete, resulting in a network of interconnected partial machines, input encoders and output decoders. An example decomposition structure is shown in Figure 1.4.

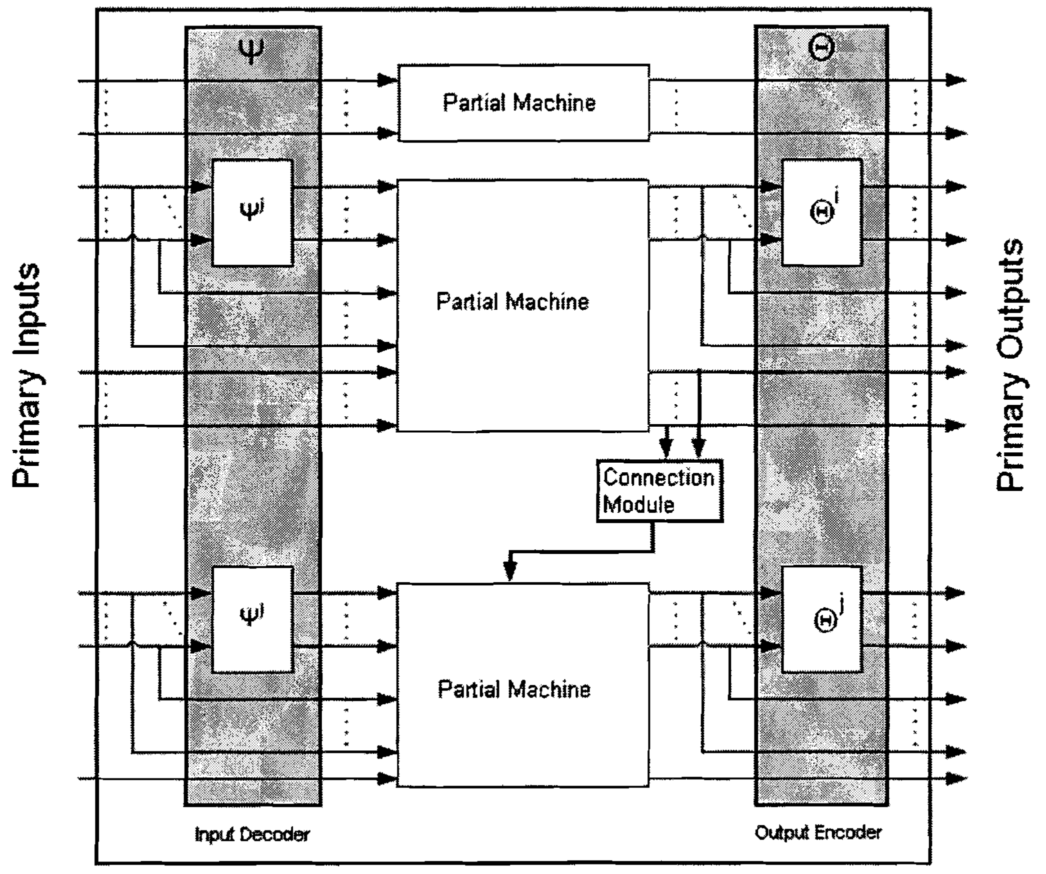


Figure 1.4: General decomposition network.

It can be seen from this figure that the actual decomposition network implementing the original FSM consists of some partial machines that can be interconnected or stand alone. Also each partial machine can have direct access to some primary inputs, or can acquire their needed input information via one or more input encoder(s). A partial machine can produce some output information. This could be enough to compute one or more output(s) directly. The remaining outputs are constructed combining output information from different partial machines using one or more output decoders.

1.4 The subject and aims of the reported work

The subject of the work presented in this report is the FSM information flow analysis for general decomposition (as stated by the title of this report and my master graduation project). This relates to the first part of the proposed FSM decomposition method. This also accounted for the final adaptation of the software developed for the second (clustering) part of the method to seamlessly collaborate with the first part.

The first aim was to develop several new or modified better algorithms for the FSM information flow structure analysis. This has been performed by Dr. L. Jóźwiak in collaboration with D. Gawlowski and myself. It was realized through reconsidering the choices made in the previous version of the algorithms (from SeMaDe) and making a much better use of all the information that is available to generate as good molecules as possible by the limited computation resource usage (memory, CPU time, etc.)

The second aim was to implement the new or improved algorithms in software (as C++ programs) and analyze if the improved version of the prototype CAD-tool works correctly and generates high-quality decompositions. The previous version of this tool (SeMaDe) is based on the logic synthesis library. This library was reused and extended whenever required. Also the adequate parts of the SeMaDe main code were reused. This way, a lot of tedious work (of writing and testing code) was avoided so that we only focused on the improvement and implementation of the actual changed and new algorithms. The so created improved version of the prototype SeMaDe CAD-tool has been thoroughly tested by performing a series of experiments with the method and the tool (with a number of benchmarks). In this phase, some bugs were found and removed. Also more efficient implementations were considered (requiring less computing resources).

The final aim was to analyze the SeMaDe method and tool, to see if they actually work correctly and generate a good quality initial set of molecules (by this is meant, a decomposition as a human engineer would do or possibly even better), that will be the base of the further synthesis, and can lead to / result in good final decompositions.

Summarizing: the aim is to have, at the end of this master project, a method based on general decomposition for generating an initial decomposition into molecules (analysis phase) with several new or improved algorithms, using as much information as possible to make the adequate decompositions by an acceptable use of computation resources. The new and improved algorithms have been implemented as part of a prototype CAD-tool (in C++) and were thoroughly tested. The method as well as its improved implementation will be described in this report, when focusing on its first FSM information flow analysis part. Finally, the analysis part was seamlessly merged with the synthesis phase of the complete tool (clustering).

2. Theoretical background

This chapter of the report will recall some selected theoretical concepts and theorems from several earlier published papers, reports and Ph.D. theses that are used in the rest of this report and are necessary for its understanding. In chapter 2.1 different types of covers will be discussed. These types of covers can be used to model the information flows in a FSM. For set systems and partitions also some operations and relations are defined (set systems are the covers we will use in the method described in this thesis, partitions are only introduced to show why we have chosen set systems to model information in our method).

In chapter 2.2 information- and abstraction sets are introduced. Besides this, relations and measures on two information- (or abstraction-)sets are defined. The information relations and measures can be used to compare two portions of information.

Chapters 2.3 explains the concept of bit supports and how they are applied in logic synthesis and decompositions of sequential machines.

Chapter 2.4 gives the definition of a sequential machine / Finite State Machine (of Moore and Mealy type). Also realizations of output and / or state behavior (of FSMs) and their necessary conditions are defined. In this chapter also set system pair algebra (and the M and m operators) are introduced.

Chapters 2.1 to 2.4 are the basis for the decomposition theory (applied to the decomposition of sequential machines). The decomposition theory states the sufficient and necessary conditions for a valid decomposition, and will be discussed in chapter 2.5.

2.1 Different types of Covers

In this paragraph the different types of covers are defined. The covers are a mathematical concepts to model information on elements of a given set S (they group together certain symbols in the set). As it can be seen in Figure 2.1 there are different types of covers: covers, unique-block covers, set systems and partitions, in order of increasing strictness. This is the classification as it has been introduced by Dr. Jóźwiak et. al. [7], [17], [22], although the notions of partitions, set systems and covers existed earlier.

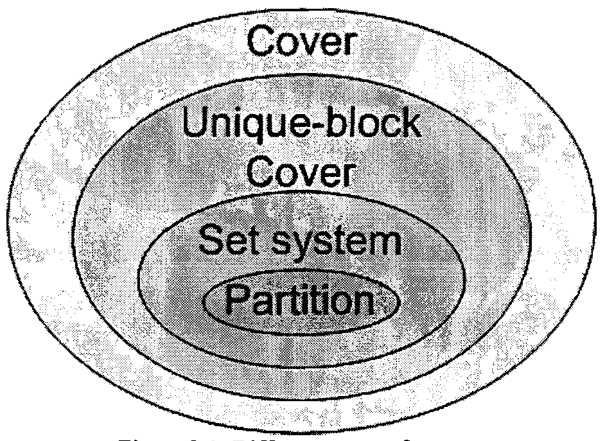


Figure 2.1: Different types of covers.

Because partitions are the most strict type of covers, they have a number of desirable properties (e.g. they form a lattice). But they have the disadvantage that they cannot be used to describe don't cares (either in the description of a FSM or Combinational function or in their decompositions). Therefore in the rest of this thesis mostly set systems will be used. Set systems do allow for don't cares but they still have many of the desirable properties of partitions (although not all if them). The (unique-block) covers can also be used to model redundancy, by using set systems in our method we cannot model redundancy, but this disadvantage is outweighed by all the other properties of set systems.

2.1.1 Definitions and Examples

The theory of general full decomposition and information relationships and measures is based on information and information flows through a sequential machine. To do this, a mathematical formulation has to be given to what information actually is. The definition we use for information, is that information is the provides the ability to distinguish different (groups of) symbols in a set.

A **symbol** is an element of set. A set contains a discrete and countable number of these elements. In the case of sequential machines (in this thesis), there are three types of symbol sets:

- S, the set of states (where a symbol is a specific state of the sequential machine).
- I, the set of input combinations (this is different from the set of inputs).
- O, the set of output combinations (this is different from the set of outputs).

Symbols with particular properties and which cannot be distinguished can be grouped. Such a group of compatible symbols is called a **block**. The type of covers is described by the restrictions that hold for these blocks.

Cover:

Definition 2.1: $$\phi_S = \left\{ B_i \mid B_i \subseteq S \wedge \bigcup_i B_i = S \right\}$$

This means that cover ϕ_S consists of a number of blocks B_i that contain together (cover) the entire set S.

Example 2.1: $$\phi_S = \{B_1;B_2;B_3;B_4;B_5\} = \left\{\overline{1};\overline{1,2};\overline{2,3};\overline{4,5};\overline{4,5}\right\}$$

Unique-block cover:

Definition 2.2: $$\phi_S = \left\{ B_i \mid B_i \subseteq S \wedge \bigcup_i B_i = S \wedge i \neq j \Rightarrow B_i \neq B_j \right\}$$

The definition of a unique-block cover is essentially the same as that of a cover with the extra restriction that no blocks are allowed to be identical.

Example 2.2: $$\phi_S = \left\{\overline{1};\overline{1,2};\overline{2,3};\overline{4,5}\right\}$$

The cover in example 2.1 is not a unique-block cover because B_4 and B_5 are identical.

Set system:

Definition 2.3: $\phi_S = \left\{ B_i \mid B_i \subseteq S \wedge \bigcup_i B_i = S \wedge i \neq j \Rightarrow B_i \subsetneq B_j \right\}$

In a set system it is not allowed for a block to be entirely covered by another block (all the symbols in a block are also in another block).

Example 2.3: $\phi_S = \left\{ \overline{1,2}; \overline{2,3}; \overline{4,5} \right\}$

The (unique-)block cover in example 2.2 is not a set system because $B_2 = \overline{1,2}$ contains all the symbols of $B_1 = \bar{1}$.

Partition:

Definition 2.4: $\phi_S = \left\{ B_i \mid B_i \subseteq S \wedge \bigcup_i B_i = S \wedge i \neq j \Rightarrow B_i \cap B_j = \varnothing \right\}$

In a partition a certain symbol is not allowed in more than one block.

Example 2.4: $\phi_S = \left\{ \overline{1,2}; \bar{3}; \overline{4,5} \right\}$

2.1.2 Operations on Partitions and Set Systems

A set system can be used to model information. For this purpose it is useful to define some operators (product $\cdot$ and sum $+$) and also some relations (smaller or equal $\leq$, larger or equal $\geq$, equality $=$). In this chapter their mathematical definition as well as there intuitive meaning are introduced for set systems.

Let ϕ_S^1 and ϕ_S^2 be two set systems on symbol set S. The following **relations between two set systems** can then be defined as:

Definition 2.5a: $\phi_S^1 \geq \phi_S^2 \text{ iff } \left\{ \forall B^2 \in \phi_S^2, \exists B^1 \in \phi_S^1 : B^1 \supseteq B^2 \right\}$

Definition 2.5b: $\phi_S^1 \leq \phi_S^2 \text{ iff } \left\{ \forall B^1 \in \phi_S^1, \exists B^2 \in \phi_S^2 : B^1 \subseteq B^2 \right\}$

Definition 2.5c: $\phi_S^1 = \phi_S^2 \text{ iff } \left\{ \phi_S^1 \leq \phi_S^2 \wedge \phi_S^1 \geq \phi_S^2 \right\}$

Definition 2.5d: $\phi_S^1 < \phi_S^2 \text{ iff } \left\{ \phi_S^1 < \phi_S^2 \wedge \phi_S^1 \neq \phi_S^2 \right\}$

Definition 2.5e: $\phi_S^1 > \phi_S^2 \text{ iff } \left\{ \phi_S^1 > \phi_S^2 \wedge \phi_S^1 \neq \phi_S^2 \right\}$

The intuitive meaning of $\phi_S^1 \geq \phi_S^2$ is that ϕ_S^2 contains more or the same information than ϕ_S^1. This might be confusing because of the definition of largest set system. The meaning of the other operations for comparing set systems are then obvious.

Example 2.5a: $\left\{\overline{1,2,3};\overline{2,3,4}\right\} \geq \left\{\overline{1,2};\overline{2,3};\overline{3,4}\right\}$

Example 2.5b: $\left\{\overline{1,2};\overline{3,4}\right\} \leq \left\{\overline{1,2,3};\overline{1,3,4};\overline{2,4}\right\}$

Again, let ϕ_S^1 and ϕ_S^2 be two set systems on symbol set S. The product of two set systems can then be defined as:

Definition 2.6a: $\phi_S = \phi_S^1 \cdot \phi_S^2 = \left\{B_i \mid \exists B^1 \in \phi_S^1, \exists B^2 \in \phi_S^2 : B_i = B^1 \cap B^2 \wedge i \neq j \Rightarrow B_i \subsetneq B_j\right\}$

The second constraint ($i \neq j \Rightarrow B_i \subsetneq B_j$) can also be seen as the Max operator, in this way the definition becomes:

Definition 2.6b: $\phi_S = \phi_S^1 \cdot \phi_S^2 = Max\left\{B_i \mid \exists B^1 \in \phi_S^1, \exists B^2 \in \phi_S^2 : B_i = B^1 \cap B^2\right\}$

The intuitive meaning of the product of two set systems is that $\phi_S = \phi_S^1 \cdot \phi_S^2$ represents the combined information of ϕ_S^1 and ϕ_S^2 (which is the largest set system that contains all the information of ϕ_S^1, as well as, ϕ_S^2).

Example 2.6: $\left\{\overline{1,2,3};\overline{3,4,5}\right\} \cdot \left\{\overline{1,3,4};\overline{1,5};\overline{2,3,4}\right\} = \left\{\overline{1,3};\overline{2,3};\overline{3,4};\overline{5}\right\}$

Again, let ϕ_S^1 and ϕ_S^2 be two set systems on symbol set S. The sum of two set systems can then be defined as:

Definition 2.7a: $\phi_S = \phi_S^1 + \phi_S^2 = \left\{B_i \mid B_i \in B^1 \cup B^2 \wedge i \neq j \Rightarrow B_i \subsetneq B_j\right\}$

Or

Definition 2.7b: $\phi_S = \phi_S^1 + \phi_S^2 = Max\left\{B_i \mid B_i \in B^1 \cup B^2\right\}$

The intuitive meaning of the sum of two set systems is that $\phi_S = \phi_S^1 + \phi_S^2$ represents the combined abstraction of ϕ_S^1 and ϕ_S^2 (which is the smallest set system that contains the information that is both in ϕ_S^1 as well as in ϕ_S^2).

Example 2.7: $\left\{\overline{1,2,3};\overline{1,5};\overline{3,4}\right\} + \left\{\overline{1,2};\overline{2,3,4};\overline{3,5}\right\} = \left\{\overline{1,2,3};\overline{1,5};\overline{2,3,4};\overline{3,5}\right\}$

We will also introduce some of these concepts for partitions:

For this we introduce the notation: $s \equiv t(\pi)$ if and only if s and t are contained in the same block of π, i.e. $s \equiv t(\pi) \Leftrightarrow B_\pi(s) = B_\pi(t)$.

Now multiplication, sum and the relation $\leq$ can be defined al follows:

Definition 2.8: $\pi_S = \pi_S^1 \cdot \pi_S^2$ is the partition on S such that:
$s \equiv t\left(\pi_S^1 \cdot \pi_S^2\right) \Leftrightarrow s \equiv t\left(\pi_S^1\right) \wedge s \equiv t\left(\pi_S^2\right)$

Definition 2.9: $\pi_S = \pi_S^1 + \pi_S^2$ is the partition on S such that:
$s \equiv t\left(\pi_S^1 + \pi_S^2\right) \Leftrightarrow$ there exists a sequence in S
$s = s_0, s_1, s_2, \ldots, s_n = t$ for which either
$s_i \equiv s_{i+1}\left(\pi_S^1\right)$ or $s_i \equiv s_{i+1}\left(\pi_S^2\right)$

Definition 2.10 Once the product and sum are defined for partitions we can define the relation $\pi_S^1 \leq \pi_S^2 \Leftrightarrow \pi_S^1 \cdot \pi_S^2 = \pi_S^1 \Leftrightarrow \pi_S^1 + \pi_S^2 = \pi_S^2$.
This is true if and only if each block of π_S^1 is contained in a block of π_S^2.

Note that when $\pi_S^1 = \left\{\overline{1,2};\overline{3}\right\}$ and $\pi_S^2 = \left\{\overline{1};\overline{2,3}\right\}$ then: $\pi_S^1 + \pi_S^2 = \left\{\overline{1,2};\overline{2,3}\right\}$ regarded as set systems or $\pi_S^1 + \pi_S^2 = \left\{\overline{1,2,3}\right\} = 1$ when regarded as partitions.

Next we will introduce some concepts which are identical for both set systems and partitions.

Definition 2.11 The cardinality $|\phi_S|$ of a set system (or partition) ϕ_S is the number of blocks in that set system (or partition)

Example 2.11 $\left|\left\{\overline{1,2,3};\overline{1,5};\overline{2,3,4};\overline{3,5}\right\}\right| = 4$

Definition 2.12a The 0 set system (or partition) is the smallest possible set system (or partition) so $0 \leq \phi$ where ϕ is any set system (or partition).

Definition 2.12b A second way of defining a 0 set system (or partition) can be done by the fact that this set system (or partition) has each symbol in a separate block:
$\phi_S(0) = \left\{B_i \mid |B_i| = 1 \wedge \bigcup_i B_i = S\right\}$

Example 2.12 when $|S| = 5$ then $\phi_S(0) = \left\{\overline{1};\overline{2};\overline{3};\overline{4};\overline{5}\right\}$

Definition 2.13a The 1 set system (or partition) is the largest possible set system (or partition) so $1 \geq \phi$ where ϕ is any set system (or partition).

Definition 2.13b A second way of defining a 1 set system (or partition) can be done by the fact that this set system (or partition) has all symbols in one block:
$\phi_S(1) = \left\{B_i \mid |B_i| = |S| \wedge |\phi_S(1)| = 1\right\}$

Example 2.13 when $|S| = 5$ then $\phi_S(1) = \{\overline{1,2,3,4,5}\}$

For the 0 and 1 set systems (or partitions) the following holds:

$$1 \cdot \phi_S = \phi_S,$$
$$0 \cdot \phi_S = 0,$$
$$1 + \phi_S = 1,$$
$$0 + \phi_S = \phi_S.$$

More information on partitions, set systems and related operations can be found in [1], [3], [5], [6], [16], [17], [19], [22], [23].

2.2 Information Relationships and Measures

In the previous section (2.1.2) was already noted that set systems and partitions can be used to model information, where information is the ability to distinguish (groups of) symbols in a set. For a certain set of symbols S, for every pair of symbols $s_i, s_j \in S$ can be determined if the two symbols are distinguishable. An ability to distinguish two from each other represents the smallest amount of information and it is referred to as elementary or atomic information [7] [22]. The notions of elementary information, information sets, as well as, the theory of information relationships and measures were introduces by Dr. Jóźwiak in [7].

For a certain set system, the set of all distinguishable pairs of symbols can be written down. This is called an **information set** and is defined as follows:

Definition 2.14a: $\inf(\varphi_S) = IS\left\{\{s_i, s_j\} \mid \overline{s_i, s_j} \subsetneq B_k^{\phi_S}\right\}$

Definition 2.14b: $abs(\varphi_S) = AS\left\{\{s_i, s_j\} \mid \overline{s_i, s_j} \subseteq B_k^{\phi_S}\right\}$

So, if two particular symbols are not contained in any block of the set system, they can be distinguished, and so this elementary distinction is present in the information set. If, however there is a block that contains both symbols, distinction is not possible and the pair is put in the abstraction set.

Example 2.14 if $\phi_S = \{\overline{0,2,3,4}; \overline{1,2,5}\}$

than $IS(\phi_S) = \{0 \mid 1, 0 \mid 5, 1 \mid 3, 1 \mid 4, 3 \mid 5, 4 \mid 5\}$

and $AS(\phi_S) = \{0 \mid 2, 0 \mid 3, 0 \mid 4, 1 \mid 2, 1 \mid 5, 2 \mid 3, 2 \mid 4, 2 \mid 5, 3 \mid 4\}$

Note that $|IS(\phi_S)| + |AS(\phi_S)| = \frac{1}{2}|S| \cdot (|S| - 1)$ where $|IS(\phi_S)|, |AS(\phi_S)|$ and $|S|$ are the cardinalities (number of elements) in the information set, the abstraction set and the number of symbols that ϕ_S is defined on respectively.

For set systems the following statements hold according to their product, sum and $\geq$ relation and their corresponding information set:

$$\inf(\phi_1 \cdot \phi_2) = \inf(\phi_1) \cup \inf(\phi_2) = TI(\phi_1, \phi_2),$$
$$\inf(\phi_1 + \phi_2) = \inf(\phi_1) \cap \inf(\phi_2) = CI(\phi_1, \phi_2),$$
$$\phi_1 \geq \phi_2 \Rightarrow \inf(\phi_1) \subseteq \inf(\phi_2).$$

For partitions this is also true except for the second statement. For partitions this is:

$$\inf(\phi_1 + \phi_2) \subseteq \inf(\phi_1) \cap \inf(\phi_2)$$

This is due to the fact that not every information set can be modeled by a partition. For example on a set of 3 symbols, 1|2 can be distinguished by $\left\{\bar{1};\overline{2,3}\right\}$ and $\left\{\overline{1,3};\bar{2}\right\}$. However using these partition also other information is distinguished. $IS\left(\left\{\bar{1};\overline{2,3}\right\}\right) = \{1\,|\,2, 1\,|\,3\}$ and $IS\left(\left\{\overline{1,3};\bar{2}\right\}\right) = \{1\,|\,2, 2\,|\,3\}$. Modeling just the information needed to distinguish 1|2 requires the symbol 3 present in both blocks, thus it can not be modeled by a partition. This is the main reason for using set systems in our method.

However a problem regarding set systems is that $\phi_1 \geq \phi_2 \Rightarrow \inf(\phi_1) \subseteq \inf(\phi_2)$ is valid in one direction only. For instance, according to this definition if $\phi_1 = \left\{\overline{1,2};\overline{1,3};\overline{2,3};\overline{3,4}\right\}$ and $\phi_2 = \left\{\overline{1,2,3};\overline{3,4}\right\}$ than it should hold that they both represent different information. However the information set for both ϕ_1 and ϕ_2 = $\{1\,|\,4, 2\,|\,4\}$. So apparently, there exists more than one way to represent some information. So to make this mapping work both ways we introduce the concept of reduced set systems such that there exists only one reduced set system that represents some particular portion of information. This would make an inverse mapping possible ($\inf^{-1}$).

Definition 2.15 A reduced set system on set S is a set system ϕ on S such that for every set system ψ on S $\inf(\phi) = \inf(\psi) \Rightarrow \phi \geq \psi$. This means that the reduced set system has the largest blocks and the least number of blocks required to represent some particular information.

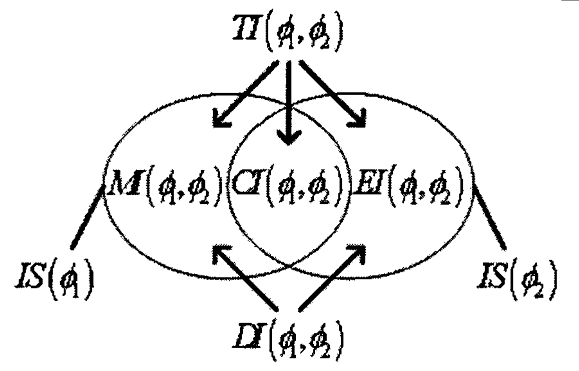
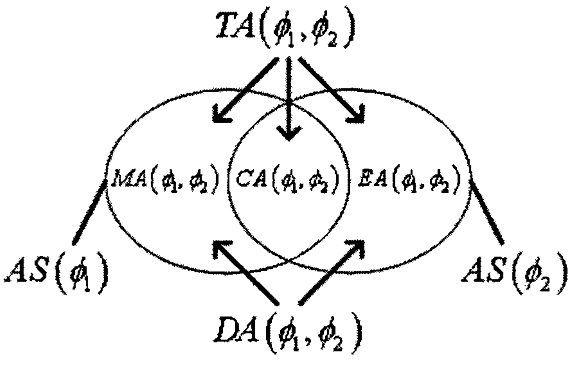
If both ϕ and ψ are reduced set systems than $\inf(\phi) = \inf(\psi) \Rightarrow \phi = \psi$. So when using reduced set systems there is only one set system representing some particular information, thus the following holds: $\phi_1 \geq \phi_2 \Leftrightarrow \inf(\phi_1) \subseteq \inf(\phi_2)$

Definition 2.16 When using reduced set systems (ϕ_1 and ϕ_2) we can also define the division operator: $\phi_1 / \phi_2 \Leftrightarrow \inf^{-1}\left(\inf(\phi_1) \setminus \inf(\phi_2)\right)$

For this division operator the following properties hold:

$$\psi \le \varphi \Rightarrow (\psi / \varphi) \cdot \varphi = \psi,$$
$$\psi \ge \varphi \Rightarrow \psi / \varphi = 1,$$
$$\neg(\psi \le \varphi \wedge \psi \ge \varphi) \Rightarrow \psi / \varphi = \psi / (\psi + \varphi),$$
$$(\psi / \phi) + \phi = 1,$$
$$\psi / 1 = \psi,$$
$$\psi / 0 = \psi / \psi = 1.$$

Using these information and abstraction sets, we can also define relations. All possible relations are denoted in the table below. This is done in the form of a formula as well a by a graphical representation:

Common Information	$CI(\phi_1,\phi_2) = IS(\phi_1) \cap IS(\phi_2)$	 $TI(\phi_1,\phi_2)$ $MI(\phi_1,\phi_2)$ $CI(\phi_1,\phi_2)$ $EI(\phi_1,\phi_2)$ $IS(\phi_1)$ $IS(\phi_2)$ $DI(\phi_1,\phi_2)$ Figure 2.2a: Information relationships.
Total Information	$TI(\phi_1,\phi_2) = IS(\phi_1) \cup IS(\phi_2)$	
Missing Information	$MI(\phi_1,\phi_2) = IS(\phi_1) \setminus IS(\phi_2)$	
Extra Information	$EI(\phi_1,\phi_2) = IS(\phi_2) \setminus IS(\phi_1)$	
Different Information	$DI(\phi_1,\phi_2) = IS(\phi_1) \cap IS(\phi_2) \setminus IS(\phi_1) \cup IS(\phi_2)$	
Common Abstraction	$CA(\phi_1,\phi_2) = AS(\phi_1) \cap AS(\phi_2)$	 $TA(\phi_1,\phi_2)$ $MA(\phi_1,\phi_2)$ $CA(\phi_1,\phi_2)$ $EA(\phi_1,\phi_2)$ $AS(\phi_1)$ $AS(\phi_2)$ $DA(\phi_1,\phi_2)$ Figure 2.2b: Abstraction relationships.
Total Abstraction	$TA(\phi_1,\phi_2) = AS(\phi_1) \cup AS(\phi_2)$	
Missing Abstraction	$MA(\phi_1,\phi_2) \doteq AS(\phi_1) \setminus AS(\phi_2)$	
Extra Abstraction	$EA(\phi_1,\phi_2) = AS(\phi_2) \setminus AS(\phi_1)$	
Different Abstraction	$DA(\phi_1,\phi_2) = AS(\phi_1) \cap AS(\phi_2) \setminus AS(\phi_1) \cup AS(\phi_2)$	

Beside these relations, also measures can be introduced on information set SS_1 and SS_2:

Information and Abstraction similarity (affinity): $ISIM(SS_1,SS_2)=\lvert CI(SS_1,SS_2)\rvert$, $ASIM(SS_1,SS_2)=\lvert CA(SS_1,SS_2)\rvert$
Information and Abstraction dissimilarity (difference): $IDIS(SS_1,SS_2)=\lvert DI(SS_1,SS_2)\rvert$, $ADIS(SS_1,SS_2)=\lvert DA(SS_1,SS_2)\rvert$
Information and Abstraction decrease (loss): $IDEC(SS_1,SS_2)=\lvert MI(SS_1,SS_2)\rvert$, $ADEC(SS_1,SS_2)=\lvert MA(SS_1,SS_2)\rvert$
Information and Abstraction increase (growth): $IINC(SS_1,SS_2)=\lvert EI(SS_1,SS_2)\rvert$, $AINC(SS_1,SS_2)=\lvert EA(SS_1,SS_2)\rvert$
Total Information and Abstraction quality: $TIC(SS_1,SS_2)=\lvert TI(SS_1,SS_2)\rvert$, $TAC(SS_1,SS_2)=\lvert TA(SS_1,SS_2)\rvert$

Above only the simplest information relationships and measures were discussed. Using these measures all elementary information items are weighted equally. However they could also be weighed according to their importance, their cost of creation, their use for production, etc. Also these measures can be normalized (such a normalized measure is used for clustering affinities in our method). More information on information relationships and measures can be found in [1], [2], [7], [9], [10], [16], [17], [22].

2.3 Bit Supports

For simplicity sake, the word “bit” is use here to denote a binary variable or a value of a binary variable.

Definition 2.17. A bit pattern of size n is a series of “0”, “1” and “-“ of length n. Value “-“ represents a “don’t care” bit and a pattern containing “-“ represents two smaller patterns, the first having “0” and the second “1” in place of “-“.

A bit pattern of size n is represents an n-dimensional cube in the n-dimensional Boolean space. A bit pattern not containing any don’t-care bits is called a minimal pattern or min-term.

Example: Pattern 1-0- represents (covers) the following minimal patterns: 1000, 1001, 1100, 1101.

Two bit patterns are distinguishable (incompatible) if they differ on at least one position, and neither of them contains “-“ in this position.

Examples.

$$\left.\begin{array}{l}\text{11-0}\\ \text{01-0}\end{array}\right\}\text{ differ on position 0 (are incompatible),}$$

$$\left.\begin{array}{l}\text{1-0-}\\ \text{-0-1}\end{array}\right\}\text{ are not distinguishable (are compatatible).}$$

In the specifications and implementations of sequential machines, the minimal bit patterns represent individual symbols. Non-minimal patterns similar to blocks of partitions or set systems represent certain groups of symbols. Every bit in a bit pattern creates a two-block partition on all minimal patterns. For example, the zeroth bit of a 5-bit pattern introduces the partition in which the first block contains all minimal patterns covered by pattern 0---- and the second block contains all minimal patterns covered by pattern 1----. If we consider information represented by a complete set of (not necessarily minimal) patterns or cubes, then every bit introduces a two-block set system on the pattern. Under the complete set we understand a set that covers the whole n-dimensional Boolean space. In the previous example, the first sub-set contains all patterns having 0 or – in the zeroth bit (e.g. 0--1- or -01--), the second sub-set contains all patterns having 1 or – in the zeroth bit (e.g. 1---0 or -11-0).

Sometimes only a selected set of bits is supplied from one sub-system to another. To express this fact mathematically, we introduce the concept of a bit support.

Definition 2.18. A bit support U of a set of bits X is any subset of X $(U \subseteq X)$ that is needed or delivered to compute the information.

When a certain support is supplied from one sub-system to another, all bit patterns received by the destination sub-system can contain information only on the bit belonging to the support. In other words, each pattern is filtered in such a way that it has 0's and 1's substituted by don't cares in all bits not belonging to the support. If x is a bit pattern and u is a bit support, we write $x|_u$ to denote such a "filtered" bit pattern.

Examples: $"1-0-0"|_{\{1,2\}} = "--0--"$

$"100-0"|_{\varnothing} = "-----"$

We write $B|_u$ to denote the set of all patterns from B filtered by u.
The filtering expresses the fact that the symbols coded as bit patterns cannot be distinguished in the sub-system receiving them by bits that are not supplied to that system.

We have stated before that every single binary variable (bit) x_i introduces a two-block set system on all bit patterns (we denote this set system as $ind(\{x_i\})$). A single bit is a special case of a bit support containing only one bit. The combined information of two bits x_i and x_j is a set system resulting from the multiplication of the set systems introduced by particular bits (we denote this set system as $ind(\{x_i, x_j\})$). This information is expressed by a bit support containing those two bits. Therefore we define the $\cdot$ and $+$ operations for supports similarly to the operations on set systems: if u and v are bit supports then:

$$ind(u) \cdot ind(v) = ind(u \cdot v) = ind(u \cup v),$$
$$ind(u) + ind(v) = ind(u + v) = ind(u \cap v).$$

Because of this correspondence between bit supports and set systems, bit supports may be used in place of set systems on coded input / output symbols in the decomposition theory. Note that the empty support represents a unity set system, but the full support does not induce a zero set system – some bit pattern pairs can never be distinguished regardless the number of bits in the support (example: a bit pattern containing all don't cares cannot be distinguished from any other pattern). We will call the set system induced by the full support a minimal induced set system.

To fully integrate bit supports and set systems, we need a mapping reverse to ind: converting a bit set system to a bit support. In doing so the following problems occur:

1. Not all possible set systems are valid ones. There are some set systems that contain contradictory information. As we have said before, some bit patterns cannot be distinguished from others (e.g. all don't care bit pattern). So set systems in which such indistinguishable patterns are distinguished are contradictory. It can be easily shown that for set system φ_B on bit patterns of length n to be a non-contradictory set system, it is necessary and sufficient, to be greater of equal to a minimal induced set system on those bit patterns, or more formally, to satisfy the condition:

   $$\varphi_B = ind\left(\left\{x_1, \ldots, x_n\right\}\right)$$

   In the sequel of this report we will work only with non-contradictory set systems.

2. The standard notation of set systems is for non-contradictory set systems on bit patterns lengthy and redundant. Consider an example set system φ_B defined on all bit patterns of length 2:

   $$\varphi_B = \left\{\overline{11, 10, 01, -1, 1-, --, 0-, -0}; \overline{00, -0, 0-, --}\right\}$$

   Since we work with non-contradictory set systems only, we can save space and simplify the notation by not expressing explicitly information which is obviously included in all non-contradictory set systems. For instance, we need not to write pattern – in the simplified notation since we know it has to occur in every block of any non-contradictory set system. We have two rules which patterns need not to be explicitly shown up in a given block:

Rule 1: If there are three bit patterns in a given block that differ only in one position, the two patterns which have 0 and 1 on that position are eliminated. In the example φ_B, the first block contains 11, 10, and 1-, so 11 and 10 may be eliminated. Similarly in contains 11, 01 and -1, so 01 may be eliminated as well.

Rule 2: We may eliminate from any block every pattern that covers at least on other pattern in that block. For instance, we eliminate patterns -0, 0- and – from the second block of φ_B.

The rules 1 and 2 are independent, but to obtain the greatest reduction, rule 1 should be applied first, followed by rule 2. If rule 2 would be appliės first, we would have a set system on minimal patterns only, so there would be nothing to reduce by rule 1.
In our example φ_B may be reduced to $\varphi_B = \left\{\overline{-1, 1-}; \overline{00}\right\}$

3. Even if we restrict our attention to non-contradictory set systems only, not all set systems have direct correspondence to a unique bit support. For instance, the set systems induced by al bit supports of length 2 are:

   $$ind(\varnothing) = \left\{\overline{--}\right\} = 1,$$

$$ind(\{x_0\}) = \varphi_{\{x0\}} = \{\overline{1-};\overline{0-}\},$$
$$ind(\{x_1\}) = \varphi_{\{x1\}} = \{\overline{-1};\overline{-0}\},$$
$$ind(\{x_0, x_1\}) = \varphi_{\{x_0,x_1\}} = \{\overline{\overline{11};\overline{10};\overline{01};\overline{00}}\}.$$

None of those supports induces φ_B: $\varphi_B < 1$, $\varphi_B > \varphi_{\{x_0,x_1\}}$, φ_B and $\varphi_{\{x_0\}}$ are incomparable and φ_B and $\varphi_{\{x_1\}}$ are incomparable.

We will approximate a set system that does not have a corresponding bit support, from above and from below, that is, approximate it by the smallest-greater-than or greatest-less-than set system having direct correspondence to the support.

The first case occurs when we consider information that can be extracted from a set system in the form of a bit support, which can be further supplied to other sub-circuits. We will call it a maximal output support problem or mosp. Computationally, mosp can be easily solved: the output support contains all bits but those which distinguish some patterns that are in one block of the considered set system. For example,

$$\text{mo}\,sp(\varphi_B) = \text{mo}\,sp(1) = \varnothing.$$

The second case occurs when we consider information that must be supported in order to compute the information contained by the set system in question. We will call this a minimal input support problem or misp. This problem is much harder to solve than mosp: first, there can be more than one minimal support. Second, finding even one of them is a NP-hard problem. On the other hand, having an efficient misp procedure is essential in combining set systems and bit supports in the application of the decomposition theory for machines with coded inputs and outputs.

In the example above, $\text{m}\,isp(\varphi_B) = \{x_0, x_1\}$ and this is the only solution.

2.4 Sequential Machines

2.4.1 Definitions of Sequential Machines (FSMs)

Digital circuits can be divided into two main classes: combinational logic and sequential circuits. A combinational circuit has no memory, so the outputs of this circuit are computed as a discrete (Boolean) function of its inputs: $\lambda : I \to O$.

Definition 2.19: A combinational machine is defined as: $M = \langle I, O, \lambda \rangle$, where:

I	: A finite set of input values;
O	: A finite set of output values;
$\lambda : I \to O$	: The output function.

A combinational machine is a mathematical model of a combinational circuit.

A sequential machine (or Finite State Machine, FSM) has a memory to remember a (finite) set of states. The FSM outputs can only be calculated knowing the state of the machine (in the case of a Moore machine), or the state and the input values (in the case of a Mealy machine). To compute the next state the inputs and the current state values are needed. In a mathematical way this is stated below:

Definition 2.20: A sequential machine M is described by a quintuple: $M = \langle I, S, O, \delta, \lambda \rangle$

Mealy:		**Moore:**	
I	– finite set of input symbols	I	– finite set of input symbols
S	– finite non-empty set of states	S	– finite non-empty set of states
O	– finite set of output symbols	O	– finite set of output symbols
δ	– next state-function $\delta : I \times S \to S$	δ	– next state-function $\delta : I \times S \to S$
λ	– output function $\lambda : I \times S \to O$	λ	– output function $\lambda : S \to O$

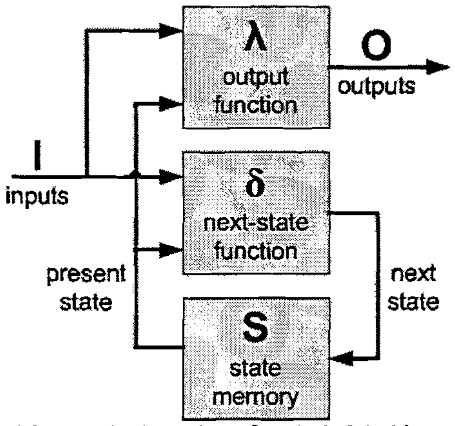


Figure 2.3a: Drawing of a Mealy Machine.

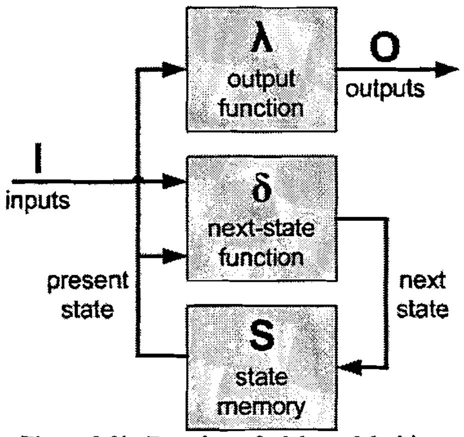


Figure 2.3b: Drawing of a Moore Machine.

As it can be seen, a sequential machine can be considered as two combinational functions (δ and λ) en some memory in which the state of the machine is stored. A combinational machine (representing a combinational circuit) can be considered as a sequential machine with only one state and a trivial next state function.

The δ and / or λ function in a sequential machine (or any other discrete function) can be completely or incompletely specified. In the case of a binary Boolean function this means that the function with n inputs is mapped to an output which can only have two values:

$B^n \to B$ where $B \in \{0,1\}$.

In the case of an incompletely specified binary function also a don't care '-' value can be assigned:

$B^n \to B^*$ where $B^* \in \{0,1,-\}$

A don't care can be used if both values (0 or 1) are possible, i.e. to represent a subset {0, 1}. This 'freedom' can and should be used by the synthesis method to create better (faster or smaller) implementations of the sequential machine / discrete function. A sequential machine is called incompletely specified if the next state and / or the output function does contain don't cares.

In this work, sequential machine specifications are assumed in which the input and outputs have already been binary encoded onto bits. So now a machine is defined by:

$$M = \langle B^n, S, D^m, \delta, \lambda \rangle$$

Where the encodings are given by the mappings:

$I \Rightarrow B^n$ and $O \Rightarrow D^m$

Which results in the next state and output function definitions:

$\delta : B^n \times S \to S$ and

$\lambda : B^n \times S \to D^m$ for the Mealy case and

$\lambda : S \to D^m$ for the Moore case.

2.4.2 Realizations of FSMs

The FSM definition as presented above:

$$M = \langle I, S, O, \delta, \lambda \rangle,$$

describes only the output and state behavior, and not how this behavior will be actually realized. Several FSMs can "simulate" or "realize" this same behavior. At least the output behavior has to be realized but sometimes it is desired to realize the state behavior as well. For a machine

$$M' = \langle I', S', O', \delta', \lambda' \rangle$$

to realize another machine M certain conditions have to be met:

$\Psi : I \to I'$, $\Phi : S \to S'$ and $\Theta : O' \to O$ (or $\Phi : 2^S \to 2^{S'}$, $\Theta : 2^{O'} \to 2^O$)

The input, state and output correspondence functions have to be defined that have to satisfy the following conditions:

For only realization of the output behavior of an incompletely specified machine:

$$\forall s \in S,\ \forall x \in I:\ \delta'(\Phi(s), \Psi(x)) \subseteq \Phi(\delta(s,x)) \wedge \Theta(\lambda'(\Phi(s), \Psi(x))) \subseteq \lambda(s,x)$$

And for realization of output and state behavior of an incompletely specified machine:

$$\forall s' \in S',\ \forall x \in I:\ \Phi'(\delta(s', \Psi(x))) \subseteq \delta(\Phi'(s'), x) \wedge \Theta(\lambda'(s', \Psi(x))) \subseteq \lambda(\Phi'(s'), x)$$

This is graphically represented in Figures 2.4a and 2.4b.

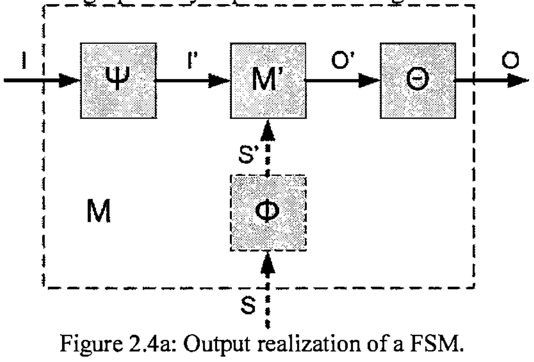


Figure 2.4a: Output realization of a FSM.

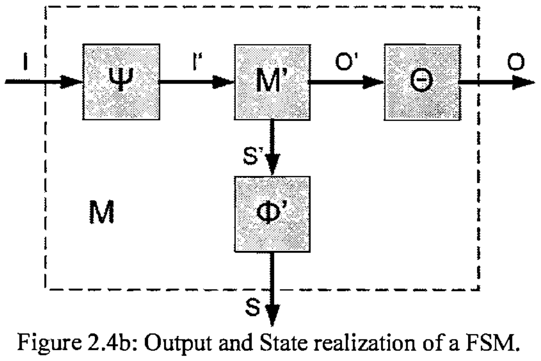


Figure 2.4b: Output and State realization of a FSM.

The reasons to be interested in FSM realizations is that the input given to our method only specifies the behavior of the machine, and the method being the subject of this report has to construct very good decompositional realizations of the specified behavior. As mentioned before, there are many ways to realize the specified behavior, even by many machines cooperating together (this is what we actually aim for with decomposition), so we can construct the best realization for our purposes.

2.4.3 Pair Algebra

In a sequential machine, there are two combinational functions $\delta(s,x)$ and $\lambda(s,x)$ for computing the next state and the, output, respectively for a particular state and input. If we do not have or require full precision regarding the input-, output- or state symbols, we define $\bar{\delta}$ and $\bar{\lambda}$ which map not input- and state-symbols but input- and state-blocks onto state- or output-blocks respectively.

These mapping functions are defined as follows:

$\bar{\delta}(B,x)=\{\delta(s,x) \mid s\in B\subseteq S\wedge x\in I\}$	$\bar{\lambda}(B,x)=\{\lambda(s,x) \mid s\in B\subseteq S\wedge x\in I\}$
$\bar{\delta}(s,A)=\{\delta(s,x) \mid s\in S\wedge x\in A\subseteq I\}$	$\bar{\lambda}(s,A)=\{\lambda(s,x) \mid s\in S\wedge x\in A\subseteq I\}$
$\bar{\delta}(D)=\{\delta(s,x) \mid (s,x)\in D\subseteq S\times I\}$	$\bar{\lambda}(D)=\{\lambda(s,x) \mid (s,x)\in D\subseteq S\times I\}$

Let $M=\langle I,S,O,\delta,\lambda\rangle$ and ϕ_S, ψ_S be set systems on S, ϕ_I be set systems on I, ϕ_O be set systems on O and $\phi_{S\times I}$ be set systems on S x I. Than the set system pairs are defined as follows:

(ϕ_S,ψ_S) is an $S\to S$ set system pair if and only if $\forall B\in\phi_S$, $\forall x\in I$: $\bar{\delta}(B,x)\subseteq B'$, $B'\in\psi_S$
(ϕ_I,ϕ_S) is an $I\to S$ set system pair if and only if $\forall A\in\phi_I$, $\forall s\in S$: $\bar{\delta}(s,A)\subseteq B$, $B\in\phi_S$
(ϕ_S,ϕ_O) is an $S\to O$ set system pair if and only if $\forall B\in\phi_S$, $\forall x\in I$: $\bar{\lambda}(B,x)\subseteq C$, $C\in\phi_O$
(ϕ_I,ϕ_O) is an $I\to O$ set system pair if and only if $\forall A\in\phi_I$, $\forall s\in S$: $\bar{\lambda}(s,A)\subseteq C$, $C\in\phi_O$
$(\phi_{I\times S},\phi_S)$ is an $I\times S\to S$ set system pair if and only if $\forall D\in\phi_{I\times S}$: $\bar{\delta}(D)\subseteq B$, $B\in\phi_S$
$(\phi_{I\times S},\phi_O)$ is an $I\times S\to O$ set system pair if and only if $\forall D\in\phi_{I\times S}$: $\bar{\lambda}(D)\subseteq C$, $C\in\phi_O$

The interpretation of the notions introduced above is as follows: (ϕ_S,ψ_S) is an $S\to S$ set system pair if and only if the blocks of ϕ_S are mapped by M into the blocks of ψ_S, i.e. the input and the blocks of ϕ_S will unambiguously determine the block of ψ_S in which the next state will be contained. In other words, knowing the input and having the information about the present state with the precision to the compatibility blocks introduced by ϕ_S, it is possible to compute the information about the next state with the precision to the compatibility sets introduced by ψ_S. The interpretation of the notions of $I\to S$, $S\to O$, $I\to O$, $I\times S\to S$ and $I\times S\to O$ are similar. Set system ϕ_S has substitution property (SP) if and only if (ϕ_S,ϕ_S) is a $S\to S$ pair (having the information about the present state with precision to the blocks of this set system, we can compute the next state information by the same precision).

Definition 2.21 Let ϕ_S be a set system on S. The minimal second set system which forms a $S \to S$ pair with ϕ_S as the first set system will be denoted by $m_{S\to S}(\phi_S)$. The maximal first set system which forms a $S \to S$ pair with ϕ_S as the second set system will be denoted by $M_{S\to S}(\phi_S)$. It can be proved that:

$$m_{S\to S}(\phi_S) = \prod_i \{\phi_i \mid (\phi_S, \phi_i) \textit{ is an } S \to S \textit{ pair}\}$$

$$M_{S\to S}(\phi_S) = \sum_i \{\phi_i \mid (\phi_i, \phi_S) \textit{ is an } S \to S \textit{ pair}\}$$

The interpretations of $m_{S\to S}(\phi_S)$ and $M_{S\to S}(\phi_S)$ set systems is the following: for a given ϕ_S, $m_{S\to S}(\phi_S)$ describes the largest amount of information which can be computed about the next state of M knowing which of the blocks of ϕ_S contains the present state. $M_{S\to S}(\phi_S)$ describes the least amount of information which must be known about the present state of M in order to be able to compute the information about the next state with precision to ϕ_S. In a strictly similar way, m and M operators are defined an interpreted for $I \to S$, $S \to O$, $I \to O$, $I \times S \to S$ and $I \times S \to O$ set system pairs.

Keeping the above in mind, we extend the M and m operators to input and output supports.
Definition 2.22 Let $M = \langle B^n, S, D^m, \delta, \lambda \rangle$ and ϕ_S be a set system on S, u be an input support, $u \subseteq \{x_0, \dots, x_{n-1}\}$ and v be an output support, $v \subseteq \{y_0, \dots, y_{m-1}\}$ then

$m_{IB\to S}\{u\} = m_{I\to S}\{ind(u)\}$	$M_{IB\to S}\{\varphi_S\} = misp(M_{I\to S}\{\varphi_S\})$
$m_{S\to OB}\{\varphi_S\} = mosp(m_{S\to O}\{\varphi_S\})$	$M_{S\to OB}\{v\} = M_{S\to O}\{ind(v)\}$
$m_{IB\to OB}\{u\} = mosp(m_{I\to O}\{ind(u)\})$	$M_{IB\to OB}\{v\} = misp(M_{I\to O}\{ind(v)\})$
$m_{I\to OB}\{\varphi_S\} = mosp(m_{I\to O}\{\varphi_S\})$	$M_{I\to OB}\{v\} = M_{I\to O}\{ind(v)\}$
$m_{IB\to O}\{u\} = m_{I\to O}\{ind(u)\}$	$M_{IB\to O}\{\varphi_S\} = misp(M_{I\to O}\{\varphi_S\})$

If ϕ, ϕ_1 and ϕ_2 are set systems on set A and ψ, ψ_1 and ψ_2 are set systems on set B then the following is true (we will write m for $m_{A\to B}$ and M for $M_{A\to B}$ to shorten notation):

I	$\varphi_1 \geq \varphi_2 \Rightarrow m\{\varphi_1\} \geq m\{\varphi_2\}$
II	$m\{\varphi_1 + \varphi_2\} = m\{\varphi_1\} + m\{\varphi_2\}$
III	$m\{\varphi_1 \cdot \varphi_2\} = m\{\varphi_1\} \cdot m\{\varphi_2\}$
IV	$\psi \geq m\{\varphi\} \Leftrightarrow (\psi, \varphi)$ is A-B pair
IX	$M\{m\{\varphi\}\} \geq \varphi$
XIII	$m\{M\{m\{\varphi\}\}\} = m\{\varphi\}$
XI	$m\{\varphi\} \leq \psi \Leftrightarrow \varphi \leq M\{\psi\}$

V	$\psi_1 \geq \psi_2 \Rightarrow M\{\psi_1\} \geq M\{\psi_2\}$
VI	$M\{\psi_1 + \psi_2\} = M\{\psi_1\} + M\{\psi_2\}$
VII	$M\{\psi_1 \cdot \psi_2\} = M\{\psi_1\} \cdot M\{\psi_2\}$
VIII	$\varphi \leq M\{\psi\} \Leftrightarrow (\varphi, \psi)$ is A-B pair
X	$m\{M\{\psi\}\} \leq \psi$
XII	$M\{m\{M\{\psi\}\}\} = M\{\psi\}$

More information on set system pairs, their algebra and applications can be found in [3], [4], [5], [6], [19].

2.5 Decomposition Theory

A general composition of n sequential machines M_i, $GC(\{M_i\},\{Con_i\})$ consist of the following objects:

1. $\{M_i = (I_i^*, S_i, O_i, \delta_i, \lambda_i), I_i^* = I_i \times I_i', 1 \le i \le n\}$, a set of sequential machines referred to as component machines.
2. $\{Con_i : \times = O_j \rightarrow I_i', 1 \le i, j \le n\}$, a set of surjective functions referred to as connecting rules of the component machines.
3. An input encoder Ψ and an output decoder Θ

Let π_I^i, π_S^i and $\pi_{S\times I}^i$ be set systems on $M = (I,S,O,\delta,\lambda)$ on I, S, and S x I, such that $\pi_{S\times I}^{ij} \ge \pi_{S\times I}^i$ and $\pi_{S\times I}'^{\,j} = \prod_{i=1,\dots,n} \pi_{S\times I}^{ij}$. Let $\pi_{S\times I} = \prod_{i=1,\dots,n} \pi_{S\times I}^i$. Let $\pi_{S\times I}^{I\ \ i}{}'$ and $\pi_{S\times I}^{S\ \ i}{}'$ be set systems induced on S x I by π_I^i and π_S^i, respectively. Let $\pi_{S\times I}^I = \prod_{i=1,\dots,n} \pi_{S\times I}^{I\ \ i}{}'$ and $\pi_{S\times I}^S = \prod_{i=1,\dots,n} \pi_{S\times I}^{S\ \ i}{}'$.
Below, the term "trinity of set systems" will be used in the sense of three strongly related set systems.

A sequential machine $M = (I,S,O,\delta,\lambda)$ has a general full-decomposition with the output behavior realization with n component machines if and only if n trinities of set systems (π_I^i, π_S^i, $\pi_{S\times I}^i$) exist, so that:

1. $\left(\pi_{S\times I}^{I\ \ i}{}' \cdot \pi_{S\times I}^{S\ \ i}{}' \cdot \pi'_{S\times I}{}^{i}{}', \pi_S{}^{i}{}'\right)$ is a $S \times I \rightarrow S$ set system pair,
2. $\pi_{S\times I}^{I\ \ i}{}' \cdot \pi_{S\times I}^{S\ \ i}{}' \cdot \pi'_{S\times I}{}^{i}{}' \le \pi_{S\times I}{}^{i}{}'$,
3. $\pi_{S\times I}^I \cdot \pi_{S\times I}^S \le \pi'_{S\times I}{}^{i}{}'$,
4. $(\pi_{S\times I}, \pi_O(0))$ is a $S \times I \rightarrow O$ set system pair,

Additional for output and state realization:

5. $\prod_i \pi_S^i = \pi_S(0)$

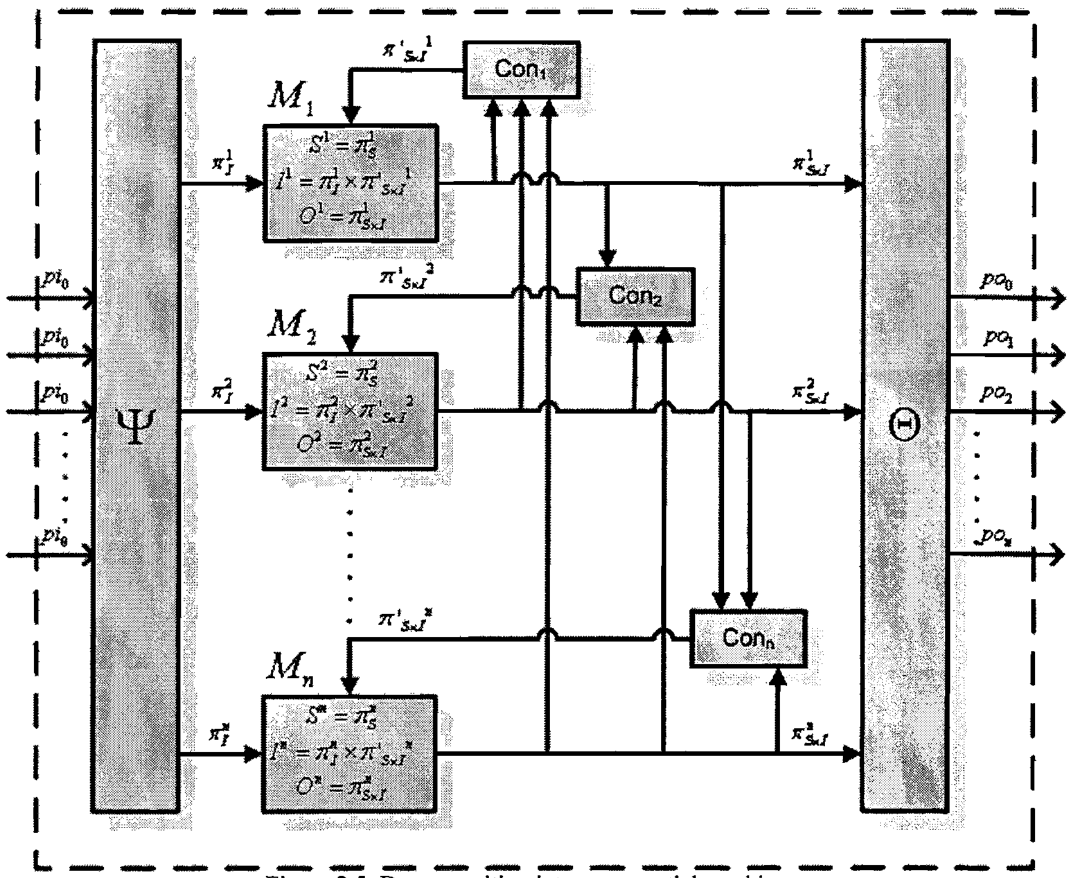


Figure 2.5: Decomposition into n sequential machines.

Intuitively these conditions can be explained as follows:

- The product $\prod_i \pi_S^i$ of the state classification relations of the partial machines forms the state classification relation $\pi_S(0)$ of the original machine (condition 5),
- The product $\prod_i \pi_{S\times I}{}^i$ of the output classification relations of the partial machines forms the classification relation which enables unambiguously computation of the output classification relation $\pi_O(0)$ of the original machine M (condition 4),
- Each partial machine is able to compute its own state and primary output classifications π_S^i and $\pi_{S\times I}^i$ based on the present state and primary input classification provided by its own state and primary input, and the classification of the elements from S x I provided by the extra input from the other machines (conditions 1 and 2),
- The composition of the partial machines is legal (condition 3).

More information on (general) decomposition theory can be found in [6], [16], [17], [19], [22].

There is an even more general theory of general full decomposition. This theory however is defined on covers. Because we use set systems to represent information in this method, we will use the theory described above, which is the most general theory of decomposition for set systems.

There is a small difference between the theoretical decompositions scheme described here and the decomposition scheme the method uses. In the theoretical decomposition scheme, each partial machine has only one output $\pi^i_{S\times I}$, the scheme in our method has two outputs β_i and ρ_i, forming together a specific case of the $\pi^i_{S\times I}$. This is because a set system on S x I would take too much memory space, therefore it is immediately mapped onto output and produced state information set systems, respectively.

Moreover, during the information analysis and clustering phases, each partial machine / (macro-) molecule has its own pseudo input encoder, pseudo output decoder, input- and output support. All the input encoders and output decoders of particular partial machines together form a specific case of the global input encoder and output decoder present in the theoretical decomposition scheme. The pseudo input encoder sometimes allows for an actual input encoder to be created later on, however its main aim is to convert the input support to input information. Something similar holds for the output decoder. The actual input encoders and output decoders are constructed during the last phase of the method (network construction) and logic synthesis. Up to that point they only exist virtually in the sense that they are allowed or suggested by the way the partial machines are created.

3 Basic data structures and operations

This chapter gives the reader basic information on the inner workings of SeMaDe. It does not give a complete overview of all its features, but its main purpose is to give a link between the theory as discussed in Chapter 2, the main algorithms that will be discussed in Chapter 4 and the way they are actually implemented in a prototype tool. By discussing the main data types and operations which can be performed to them, the main algorithms in chapter 4 can easily be described as applying the basic operations on the data structures.

3.1 Main Data structures used in SeMaDe

As mentioned, not all data structures in SeMaDe are described in this chapter. However, these are the base data structures, and the main algorithms from SeMaDe can be explained completely by the knowledge related to these data structures as presented in this chapter.

3.1.1 Supports

A support represents a sub-set of inputs related and / or supplied to a certain piece of FSM logic (a.k.a. input bit partition) or outputs produced by a particular piece of logic and supported to the total output (a.k.a. itput bit partition). Support is implemented as a vector of "bit" type to achieve easy and fast member functions (instead of a 0-1 vector, what would be sufficient). However, only "relevant" and "irrelevant" values of "bit" are used. Standard constructor requires the number of input bits to be given. It creates the full support (all bits in the support).

Of course, functions have been implemented to perform the + **operation** or union of to supports. This is implemented by an or function on both supports. Also, the **x function** on concatenation vs. augmentation which returns the joined support.

For direct manipulation on bits, the adding removing and toggling of bits, the **addBit**, **delBit** and **toggleBit** functions are implemented. Also, **makeFull** and **makeEmpty** functions are available for fast creation of the full support (zero partition) and the empty support (equivalent to a one-block partition), respectively.

The **contains** function checks if the given bit belongs to the support. **IsFull** and **isEmpty** check if the support is a full or an empty support. The **length** function returns the actual number of (relevant) bits in the support and **numBits** returns the number of bits on which the support is defined.

The **reduced** function renders for all bits not in the support the corresponding variables don't care in the given cube or "irrelevant" in a given output cube. And the **distinguishable** function checks if for this support, the given cubes are still distinguishable which is equivalent to Cube::distinct(reduced(T1), reduced(T2)) but faster.

Sing_dist returns the number of a bit in the support that distinguishes T1 and T2 and Returns -1 if more than one or no such variable exists. **Mult_dist** returns a reduced support, containing only those bits from the support that allow for distinguishing T1 and T2.

Finally some relations and operations on supports are implemented. The relations are implemented by a **compare** function which returns -1 if S1 < S2, 0 if S1 == S2, 1 if S1 > S2 and 2 if S1 and S2 are incomparable.

The following operators are implemented: **complement** (or ~) this complemented support contains the bits that weren't in the support and those that where in it are removed. The **sum** (+) returns the gives common support (a sum of bit partitions). And finally, **multiplication** (*) returns the joined support (a product of bit partitions).

3.1.2 Set Systems

The set system is the main data structure for information processing in SeMaDe. Its name is a little confusing because it is actually implemented as a set of elementary abstraction / information items, and, therefore it has more resemblance to an abstraction or information set.

More precisely, a set system is actually implemented as a bit-vector of length $\frac{1}{2}\cdot|S|\cdot(|S|-1)$ rounded upward to the nearest multiple of the word-length of the used system (usually 32). Here $|S|$ is the cardinality (number of symbols) of the set S on which the set system is defined. Because this bit-vector is only a one dimensional data structure, a function (**pairix**) has been implemented to access a particular position in this bit-vector for a given $\{s_i, s_j\}$.

Of course all the operations for set systems are discussed in Chapter 2. The multiply (*) and the addition (+) are performed by doing an or, or and operation, correspondingly, on the underlying symbol set. Division (/) is done by removing all elements in the second set if they where present in the first set and than doing the complement. Also the *= and += operations are implemented.

The relation <= returns true if the or function of the first and the second set system is equal to the first set system (meaning that the second set system contains no symbols not already present in the first set system). The == returns true if both set system actually contain exactly the same symbols.

Also there are functions implemented which check efficiently if a set system is a 0 or a 1 set system. And to change a set system into a 0 or a one set system. Using these special functions for that purpose is more efficient than doing or checking these operations for each compatibility separately.

However also all elementary information items (compatibilities) or abstraction items (incompatibilities) are also available separately to the user by the member functions **areCompatible**, **numCompatibilities**, **numIncompatibilities**, **makeCompatible** and **makeIncompatible**.

Finally, the best case bound estimate of the number of bits needed to encode this information on is returned by **numBits**. But a more precise estimate can be obtained by converting the set system to a block set or a partition. There are also member functions implemented for conversion between set systems and partitions but they are discussed in more detail in Chapter 3.2.1

3.1.3 Molecules / Clusters

A molecule or cluster is the main container of coherent portions of information. They represent the information structures that are the result of the FSM information flow analysis. As of will be discussed in Chapter 4, there are 4 different types of molecules, with two fundamentally different structures. However, they both are represented by the same data structure.

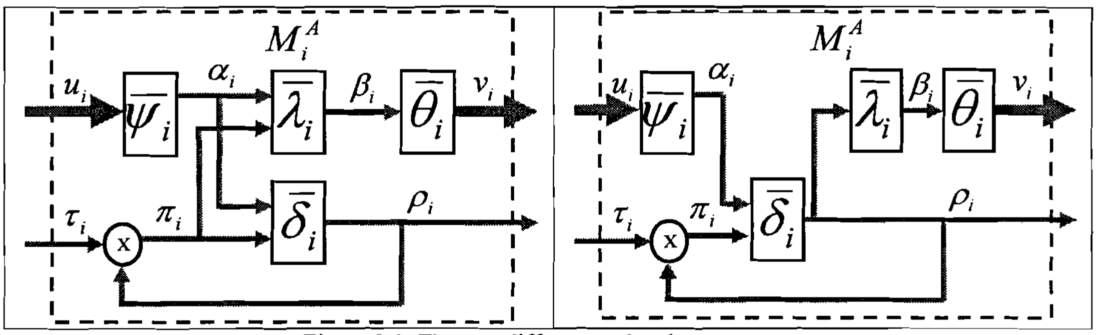


Figure 3.1: The two different molecule structures.

A molecule or cluster is define by four set systems and two supports that completely define the information flow inside and at the terminals of the molecule. These are:

- The input support u_i,
- The input information set system α_i,
- The consumed state information set system π_i,
- The output support v_i,
- The output information set system β_i,
- And the produced state information ρ_i

The imported state information τ_i is calculated or updated when needed. Of coarse, for all these supports and set systems there are get-functions available. Besides this, a molecule has a Name (stored as a string of characters) and an unique id (stored an integer) for identification. Also a molecule has a cost function calculation function which calculates its cost as will be discussed in chapter 4.

Next there are functions that return if a molecule is covered by another molecule (produces the same or more output and state information) or if a molecule is dominated (when it is covered by a molecule that requires less or the same input and state information).

Also the produce, reduce and remove output bit functions are available which force a molecule to produce some particular portion of state information, reduce its produced state information with some set system or remove an output bit respectively.

Finally, a molecule has some functions available for clustering. The merge function creates a molecule that produces at least the same amount of output and state information as its two parents. Once the output and produced state information is determined (this initially is a product of the corresponding set systems of its parents), the molecules is imploited (being forced to reduce the consumed information to the essential only) and then exploited (being forced to maximally use consumed information). Also, a member function for computing a molecule's affinity to another molecule is available.

3.1.4 Cluster Sets and Cluster Set Iterators

Clusters or molecules have to be stored in a convenient way. This is done by storing them in a cluster set. Also a cluster set iterator is available which makes it very easy to select a particular molecule from a cluster set. A cluster set is not only a container but it also has some additional functionality.

The main functionality of the cluster set is to add newly created clusters into this container for easy access and conservation. This can be done in three ways a cluster or an entire cluster set can be added, put or best put to a cluster set. For an entire solution set, also a copy function is available.

When adding a cluster to a cluster set, the cluster is always added to the set no matter what other molecules where already present at that time. When a cluster is put to the cluster set, it is only actually added when the cluster is not dominated by any other clusters in the set. Also other clusters that are dominated by the new cluster are removed. Using the best-put functionality, it is only actually added when the cluster is not covered by any other clusters in the set. Also other clusters that are covered by the new cluster are removed.

The add, put and best_put functions are also implemented for adding, putting and best_putting entire cluster sets to a cluster set. With there comparable functions for a single cluster are used, but now for each cluster in the cluster set. After these functions are done, the clusters are removed from the cluster set that they where originally in and this cluster set is then removed. Therefore also a copy function is implemented which copies an entire cluster set.

Finally there are some functions for detaching / removing a cluster from a cluster set based on the name of or a reference to the cluster, selecting a cluster from the cluster set based on its position and for requesting the cardinality (number of clusters in) the cluster set.

A cluster set iterator is implemented for easy access to the separate clusters in the cluster set.

Further I would like to mention that the SeMaDe tool has 5 main cluster sets:

1. The first one contains the A molecules
2. The second one contains the split A molecules
3. The third one contains the Moore molecules
4. The fourth one contains the B molecules
5. And the fifth one contains all the clusters created during clustering by merging.

These five cluster set together contain all molecules created by SeMaDe during a decomposition run. Besides this each solution also contains two cluster sets (see chapter 3.1.5), a flagged and a non-flagged cluster set, however they do not contain any clusters that are not present in one of the five main cluster sets.

3.1.5 Solutions

A solution is a data structure that mainly consists of two cluster sets. One being a flagged cluster set and the other a non-flagged cluster set. This distinction is made for clustering. Molecules that are already completely finished after the analysis phase (because it is obvious that no further improvement is possible or only leads to worse solutions) are not changed by clustering. By putting these finished molecules in a separate cluster set (flagged cluster set), the clustering only considers the molecules in the non-flagged cluster set. Another important parameter is the clustering level. This parameter is further explained in the chapter on clustering, and indicates the clustering progress of that particular solution. Finally, a solution has a cost which is the sum of the costs of all the clusters in the flagged set as well as the non-flagged set.

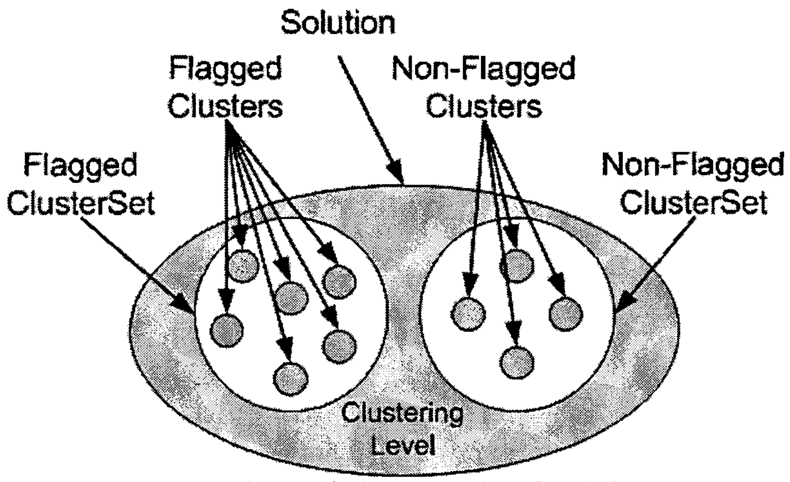


Figure 3.2: Graphical representation of a solution.

3.1.6 Solution Sets and Solution Set Iterators

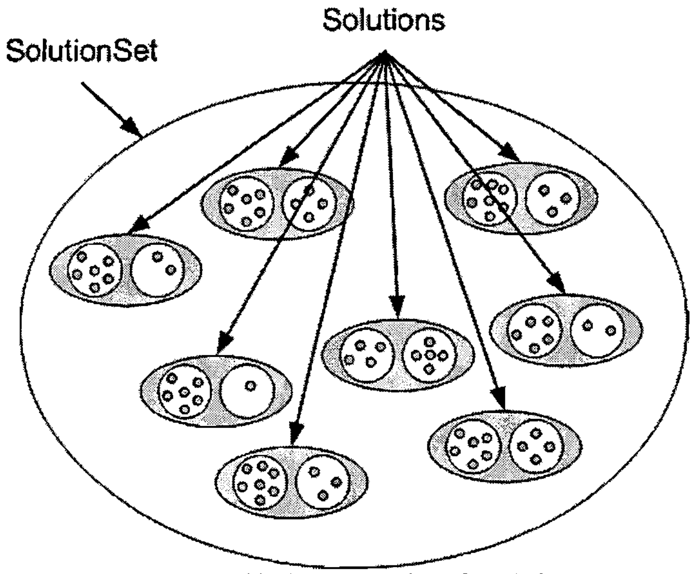


Figure 3.3: Graphical representation of a solution set.

A solution set is a list of solutions. Solutions or other solutions sets can be added or put to an existing solution set. The difference between the add and the put function is that when using the add function, the solution is always added to the solution set. Using the put function, the solution is only added to the solution set when in is not already present. It could already be present because a same solution can be created trough different paths of the clustering algorithm.

A detach function is available to remove solutions from the solution set and a cardinality is present which returns the number of solutions in the set. These functions are mainly used by the second beam of the clustering algorithm (see Chapter 4).

Finally a solution set has a cluster set which contains references to all clusters / molecules created during clustering by merging molecules. This is done because memory leakage can occur when references are lost during the removal of solutions during the second beam of the clustering algorithm. This can also be solved to check every when a solution is removed if some cluster is removed that is not present in any remaining solutions, however this is a cumbersome process which is prone to errors.

Also a solution set iterator is implemented which makes it very easy to select a particular solution from the solution set by iterating over the solution set.

3.2 Basic operations and algorithms

In this chapter related to basic partial algorithms, some basic operations and algorithms required for FSM information analysis are discussed. Many of these operations where already introduced in the Chapter about the theory of information analysis, where their meaning and use in decomposition and specifically in information analysis was clarified. However, here we focus more on how these operations are actually performed using a computer. Also their complexity and required resources in terms of memory and time are discussed. By explaining these basic operations in detail first, we can simply refer to the specific operation when discussing the main algorithms.

3.2.1 Conversions between information representations

SeMaDe uses a number of data types for storing information. First, I would like to clarify the names of these types which are a little confusing because of the history of SeMaDe.

The main data structure for the representation is the Set system (see also the chapter on data structures). The name of this data structure is confusing because it is actually implemented like an information set (a set of elementary information items) corresponding to a given set system. All partial algorithms as discussed further in this chapter operate on this data structure. However in some cases it is useful to have a different representation of the same or almost the same information.

A partition is also implemented as the data structure partition. It is represented as a characteristic function. A characteristic function assigns to each symbol a number which identifies the block that this symbol is in. This is a very compact representation, which is also fast for operations. However, using this representation, a particular symbol can only be present in one block. This is fine for partitions because as discussed in the chapter on the theory, in a partition a symbol can only be present in just one block. In Set Systems however, this is not the case. Set Systems do have a characteristic function but now a symbol can be assigned to more than one block, so there are no advantages in terms of speeding up operations or ease of use to this way of representation.

As mentioned earlier, in the SeMaDe program, the main way of representation is the Set System (which is actually implemented as an information set). However this representation of information has one serious disadvantage which is that there is no (easy) way to get information on the actual structure of the represented information. Therefore, an additional data structure was created which actually resembles the representation of an actual set system. This representation is implemented as a group of blocks, which in turn, is a group of symbols.

Although most of the information processing in SeMaDe is done on information represented as an information set (called set system in SeMaDe). Conversions from and to partitions and block sets (set system) were made available.

Information Set -> partition (upper and lower approximation)

As mentioned in Chapter 2 on the theory, a partition can not be used to model don't care. So not every set system (those which have symbols in more than one block), cannot be converted to a partition containing exactly the same information as the set system. So there are two approximations of set systems by a partition that are as similar to the set system as possible, the upper and the lower approximation.

The upper approximation converts the set system to the smallest possible partition containing no more information than the set system. This is a computationally easy procedure and this is done by starting with a 0 partition (all information present), and merging the pairs of symbols which are also compatible in the set system (information set representation).

The lower approximation converts the set system to the greatest possible partition containing no less information than the set system. This problem is equivalent to Graph-Coloring which is an NP-hard problem. A graph coloring problem is the problem where the nodes of a graph need to be colored with as few colors as possible, however two nodes with an edge between them may not have the same color. In this case each node represents a symbol and an edge between two nodes in the graph means that they are incompatible. The color of each node identifies the block that that particular symbol is in. Each node can only have one color, so the result of coloring will always be a partition. And every pair of nodes with an edge between them must have a different color so at least all the in formation in the set system (information set) is at least contained in the partition. As mentioned, the problem is NP-hard, so it is solved by a heuristic coloring algorithm. In our implementation we use a variant of the color influence method. Because this is a heuristic algorithm it could be (especially for larger instances of the problem) that more colors that strictly needed are required.

Partition -> Information Set

The conversion from a partition to a set system is a fairly easy procedure because for every partition a set system (information set) can be found that represents exactly the information contained in a partition. This is simply done by checking for every pair of symbols in the partition if they are in the same block. If this is the case they should also be compatible in the set system (added to the abstraction set).

Information Set -> Set System (clique splitting algorithm)

The conversion of a set system (represented as an information set) to a block set (theoretical set system) is a problem of finding a complete covering of a graph of as few and as large cliques (which are fully connected sub-graphs) as possible. The original implementation of this maximal clique covering was of exponential complexity. This algorithm was implemented by starting with the abstraction graph (where the nodes represent the symbols and the edges represent a compatibility between two symbols), and larger and larger cliques where constructed (bottom up) until the largest cliques where found. This is in deed exponential in the number of symbols because by adding one symbol, the number of possible cliques doubles. The runtime would be unacceptable for set systems with more than about 25 symbols (over 15 min) while easily set systems of more that 2000 symbols can occur.

Therefore, I developed and implemented a new algorithm which is actually in the worst case low order polynomial in the number of symbols. This is done by starting from the different side of the problem. We assume, that we have a 1 set system (no information) and we do not cluster according to abstraction but we split according to information. This has the advantages that only one edge has to be missing in order to split the blocks of a set system (in stead of having to find a fully connected sub-graph). Secondly the final solution (a set system with only few but large blocks) will be much closer to the starting point. This algorithm makes it possible to convert an information set to a **reduced** set system in very reasonable time (about a few milliseconds even for set systems of more that 1000 symbols). This makes it possible for our method and tool to not only analyze the actual information in self but also analyze its structure (number of bits needed to encode the information on). This algorithm is used in the information analysis part and the clustering part to calculate the cost function and to print the set system (in block set representation) in the log file.

The algorithm works as follows.

1. Start with the one set system φ on set S (cardinality is known by the different number of symbols from the information set IS).
2. Iterate over all the atomic distinctions $s_i \mid s_j$ in the information set IS (independent of the exact order of iteration).
3. Iterate over all blocks of the φ set system. (independent of the exact order of iteration).
4. If a block contains both s_i and s_j split this block in two block identical to the original but one with s_i and the other with s_j removed.
5. Put both blocks back into the set of block of φ. Here put means that a new block can cover smaller blocks already present in φ but it can also be covered by a larger block in φ. A block B_1 is covered by a different block B_2 ($B_1 \leq B_2$) if B_2 contains at least all the symbols that are in B_1.

This algorithm is clarified by the following example:
Let e.g. be: $IS = \{0 \mid 3, 0 \mid 4, 1 \mid 3, 2 \mid 5, 3 \mid 5\}$

Since this information set is defined on S with $|S| = 6$, the initial $\varphi = \{\overline{0,1,2,3,4,5}\}$.

The first distinction considered is: $s_i \mid s_j = 0 \mid 3$. This will result in a new $\varphi = \{\overline{0,1,2,4,5}; \overline{1,2,3,4,5}\}$. None of the new blocks was covered because there was only one block which was split in this step. Then $s_i \mid s_j = 0 \mid 4$ leads to: $\varphi = \{\overline{0,1,2,5};\overline{1,2,4,5};\overline{1,2,3,4,5}\}$ because only the first block contains both 0 and 4. Hover the new second block is covered by the remaining original block so $\varphi = \{\overline{0,1,2,5};\overline{1,2,3,4,5}\}$. Note that adding information does not necessarily mean that more blocks are needed to represent this information. Adding information can actually decrease the number of blocks (also removing information can increase the number of blocks). Adding $s_i \mid s_j = 1 \mid 3$ leads to: $\varphi = \{\overline{0,1,2,5};\overline{1,2,4,5};\overline{2,3,4,5}\}$. Adding $s_i \mid s_j = 2 \mid 5$ leads to: $\varphi = \{\overline{0,1,2};\overline{0,1,5};\overline{1,2,4};\overline{1,4,5};\overline{2,3,4};\overline{3,4,5}\}$ so all the blocks where split with no block covering an other block. Finally adding $s_i \mid s_j = 3 \mid 5$ leads to: $\varphi = \{\overline{0,1,2};\overline{0,1,5};\overline{1,2,4};\overline{1,4,5};\overline{2,3,4};\overline{3,4};\overline{4,5}\}$. And when removing the covered blocks: $\varphi = \{\overline{0,1,2};\overline{0,1,5};\overline{1,2,4};\overline{1,4,5};\overline{2,3,4};\}$. So adding $s_i \mid s_j = 3 \mid 5$ results actually in a decrease of the number of blocks. Note that $|\varphi| = 5$ so 3 bits are required to encode this information. Note that adding $s_i \mid s_j = 1 \mid 2$ results in $\varphi = \{\overline{0,2};\overline{0,1,5};\overline{1,4,5};\overline{2,3,4};\}$ which can be encoded only on two bits. Which results in a major improvement regarding transportation of this information.

Note that blocks are only split when this is essential to distinguish some elementary portion of information. Therefore, the number of blocks in the final set system is minimal. Also their size is as large as possible since no splitting of blocks is performed unless this is needed to represent some particular elementary information. This way, it can be proven that this always is a reduced set system.

Set System -> information set

Converting a block set (set system structure) back to a set system (information) is again fairly straight forward. This is simply done by iteration over all blocks from the set system (block set) and for each particular block add each pair of symbols included in that block to the abstraction set.

3.2.2 Induced information (by inputs or outputs)

Also the induced information (indi: induced by input and indo: induced by output)operations are fairly straight forward. Below the code for the induced by input function is presented. The code for the induced by output function (indo) can be found by replacing **Inputsup**, **numInputSymbols**() and **inputCode**(symbol) by **outputsup**, **numOutputSymbols**() and **outputCode**(symbol) respectively.

```
SetSystem& Machine::indi( const Support& inputsup ) const // IB -> I
{
  SetSystem &ss = *new SetSystem( numInputSymbols() );
  for (symbol a = 0;  a < numInputSymbols()-1;  a++)
  {
    Cube& A = inputCode(a);
    for (symbol b = a+1;  b < numInputSymbols();  b++)
    {
      Cube& B = inputCode(b);
      if ( !inputsup.distinguishable(A, B) ) ss.makeCompatible(a, b);
    }
  }
  return ss;
}
```

This code works as follows. First a new 0 set system is created (containing all information). Then for each pair of symbols is checked if they are compatible. Because the symbols actually represent a cube (combination of input or output bits) they have to be looked up in the symbol dictionary by the **inputCode** and **outputCode** functions. These functions return the cubes represented by the particular symbol. If these cubes cannot be distinguished (for an explanation on this, I refer to chapter 2.3) the symbols should also be made compatible in the set system.

3.2.3 Minimal input support and Maximal output support

The minimal input support problem can be represented as a row covering problem, for which a matrix is constructed, where each row represents an atomic portion of information to be covered and the columns represent the input bits. If an input bit can distinguish a particular portion of information a one should be placed at its particular place in the matrix (otherwise a zero). The row covering problem tries to find a subset of inputs (columns) as small as possible that cover all required information (rows). Because of this equivalence to row covering, the misp problem is NP-hard.

In SeMaDe this problem is solved in two ways, by a best first search and a QuickScan (both heuristic). For both methods this covering matrix should be constructed as described above. Also between iterations of both methods, the matrix is preprocessed for the next iteration (e.g. removing already covered information and adding essential columns to the solution). In the end the results of both methods are compared and the input support from the best method is selected.

The best first search simply adds essential inputs to the minimal input support (some information can only be covered by one particular input). Then, other inputs are added that at that iteration cover the most remaining information.

The QuickScan is a smarter (but still heuristic) algorithm developed by Józwiak and Konieczny [8], [15], [19], [20]. *QuickScan* performs a quick traverse through several local minima (non-redundant supports). It works in two modes: the down-mode, when the algorithm seeks a leftmost (according to the lexicographic order) local minimum and the up-mode, when the algorithm escapes from the local minimum in the rightmost direction as can be seen in Figure 3.1. Here the grey nodes represent infeasible solutions, black arrows – the downward moves to the leftmost direction, grey arrows – the upward moves to the rightmost direction. In the up-mode, the algorithm constantly tries to discover a new local minimum, before it makes the next step upwards. In result, it walks on the surface consisting of patterns representing valid supports, located in the search space just above the patterns that do not represent valid supports (thus on the surface representing the non-redundant valid supports). In order to prevent revisiting a previous local minimum, the algorithm considers only those supports that are on the right of the previously visited supports. The best local minima visited constitute the result of the algorithm. Although the global minimum is not guaranteed, the deeper a local minimum is, the more possible ways lead to it. Therefore, the chance is little that the algorithm will skip a solution that is much better than the best visited so far. The pseudo-code of the algorithm is given below.

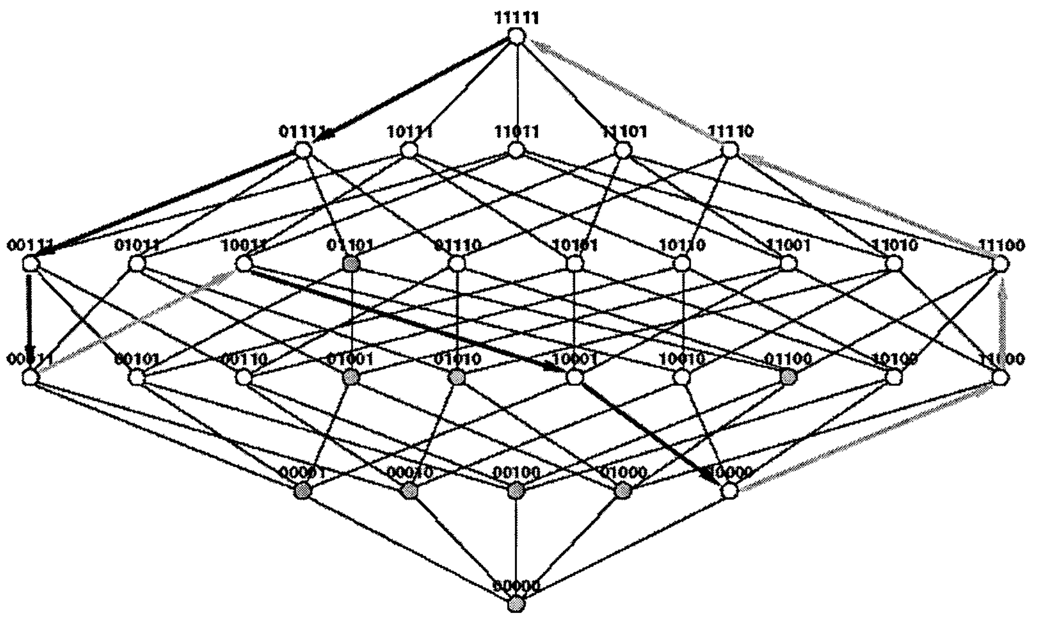


Figure 3.4: The search space and the scan path of the QuickScan algorithm.

```
procedure QuickScan
{
        current support = full support;
        best support = none;
        while (current support exists)
        {
                if (last move was down or first move)
                {
                        next support = leftmost down feasible from
                        current support;
                        if (next support exists)
                        {
                                current support = next support;
                        }
                        else
                        { // local minimum
                                if (current support < best support)
                                {
                                        best support = current support;
                                }
                                current support = rightmost up from current support;
                        }
                }
                else // last move was up
                {
                        next support = leftmost down feasible but to the
                        right from previous support;
                        if (next support exists)
                        {
                                current support = next support;
                        }
                        else
                        {
                                current support = rightmost up from current support;
                        }
                }
        }
        return best support;
}
```

As mentioned in the chapter on the theory, mosp is much easier to compute. For this, first an empty support is created with the size of the number of output bits. Then for all compatible pairs of symbols in the abstraction set (ss) it is checked what input bits can distinguish the cubes represented for these two symbols. The final returned result is the union of all these supports. The actual source code is shown below.

```
Support Machine::mosp( const SetSystem& ss ) const
{
  Support u( numOutputBits() );
  for (symbol a = 0;  a < numOutputSymbols()-1;  a++)
  {
    Output& A = outputCode(a);
    for (symbol b = a+1;  b < numOutputSymbols();  b++)
    {
      if ( ss.areCompatible(a, b) )
      {  // different bits in A and B cannot be in the support
         Output& B = outputCode(b);
          u = u + ~u.mult_dist(A, B);
      }
    }
  }
  return u;
}  // Machine::mosp
```

3.2.4 The m and M operators

In the chapter on the theory, the m and M operators where defined as respectively the product or sum of all set system pairs. This definition is unfortunately not a good way for implementing these operators.

The algorithm for computation of the $m_{S \to S}$ starts with a 0 set system (containing all information) called res, which will finally contain all next state information that can be computed from present state information p. Then, we iterate over all compatible pairs of symbols (elementary abstraction items) from the p set system. For each such compatible pair, we assume that we know all input information, so we can also compute all the pairs that are mapped onto res by the next state function delta compatible. Below, the code for the $m_{S \to S}$ operator is shown. The code for the $m_{I \to S}$ is actually quite similar only now it not iterated over the compatible pairs of states, but the compatible pairs of input symbols. $m_{S \to S}$ is quadratic in the number of states and linear in the number of input symbols, for $m_{I \to S}$ it is the other way around.

```
SetSystem& Machine::m_S_S( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol s = 0;  s < numStates()-1;  s++)
    for (symbol t = s+1;  t < numStates();  t++)
      if ( p.areCompatible(s, t) )
        for (symbol i = 0;  i < numInputSymbols();  i++) res->makeCompatible( delta(s, i), delta(t, i) );
  return *res;
}
```

```
SetSystem& Machine::m_I_S( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol i = 0;  i < numInputSymbols()-1;  i++)
    for (symbol j = i+1;  j < numInputSymbols();  j++)
      if ( p.areCompatible(i, j) )
        for (symbol s = 0;  s < numStates();  s++) res->makeCompatible( delta(s, i), delta(s, j) );
  return *res;
}
```

Below, the code for the $m_{S\to O}$ and the $m_{I\to O}$ operator is shown. This code is quite similar to that of $m_{S\to S}$ and $m_{I\to S}$ respectively. The main differences are that in case of a Moore machine, the result can be computed very fast. Secondly the mapping for these operators is done by the output function lambda (in stead of the next state function delta). $m_{S\to O}$ is quadratic in the number of states and linear in the number of input symbols, for $m_{I\to O}$ it is the other way around.

```
SetSystem& Machine::m_S_O( const SetSystem& p ) const
{
  if ( !zero_output_part ) ((Machine*)this) -> zero_output_part = &indo( Support( numOutputBits() ));
  SetSystem *res = new SetSystem( *zero_output_part );
  for (symbol s = 0;  s < numStates()-1;  s++)
   for (symbol t = s+1;  t < numStates();  t++)
     if ( p.areCompatible(s, t) )
       for (symbol i = 0;  i numInputSymbols();  i++) res->makeCompatible( lambda(s, i), lambda(t, i) );
  return *res;
}
```

```
SetSystem& Machine::m_I_O( const SetSystem& p ) const
{
  if ( !zero_output_part ) ((Machine*)this) -> zero_output_part = &indo( Support( numOutputBits() ));
  SetSystem *res = new SetSystem( *zero_output_part );
  if ( isMoore() ) return *res;
  for (symbol i = 0;  i < numInputSymbols()-1;  i++)
   for (symbol j = i+1;  j < numInputSymbols();  j++)
     if ( p.areCompatible(i, j) )
       for (symbol s = 0;  s < numStates();  s++) res->makeCompatible( lambda(s, i), lambda(s, j) );
  return *res;
}
```

For the $m_{I\times S\to S}$ and $m_{I\times S\to O}$ operators, set systems are defined on the input information as well as the output information. So there is iterated over all pairs of compatible input and state information. So these operators are quadratic in the number of inputs as well as the number of states. The code for these operators is shown below.

```
SetSystem& Machine::m_IxS_S( const SetSystem& x, const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );

  for (symbol i = 0;  i < numInputSymbols();  i++)
   for (symbol j = i;  j < numInputSymbols();  j++)
    for (symbol s = 0;  s < numStates();  s++)
     for (symbol t = s;  t < numStates();  t++)
      if ( x.areCompatible(i, j) && p.areCompatible(s, t) )
      {
        res->makeCompatible( delta(s, i), delta(t, j) );
        res->makeCompatible( delta(s, j), delta(t, i) );
      }
  return *res;
}
```

```
SetSystem& Machine::m_IxS_O( const SetSystem& x, const SetSystem& p ) const
{
  if ( !zero_output_part ) ((Machine*)this) -> zero_output_part = &indo( Support( numOutputBits() ));
  SetSystem *res = new SetSystem( *zero_output_part );
  if ( isMoore() ) return *res;
  for (symbol i = 0;  i < numInputSymbols();  i++)
   for (symbol j = i;  j < numInputSymbols();  j++)
     for (symbol s = 0;  s < numStates();  s++)
       for (symbol t = s;  t < numStates();  t++)
        if ( x.areCompatible(i, j) && p.areCompatible(s, t) )
        {
          res->makeCompatible( lambda(s, i), lambda(t, j) );
          res->makeCompatible( lambda(s, j), lambda(t, i) );
        }
  return *res;
}
```

$m_{IB\to S}\{u\}$, $m_{S\to OB}\{\varphi_S\}$, $m_{IB\to OB}\{u\}$, $m_{I\to OB}\{\varphi_S\}$ and $m_{IB\to O}\{u\}$ are actually calculated the same way as described in the chapter on the theory as a combination of the ind (induced by input or output), the mosp (maximal output support problem) and corresponding their corresponding m operators.

The algorithm for computation of the $M_{S\to S}$ starts with a 0 set system (containing all information) called res, which will finally contain the minimal amount of present state information needed to compute the next state information in p.

Then, we iterate over all compatible pairs of state symbols (elementary abstraction items). Only if for all input combinations (all input information is assumed available) the pair of elementary information mapped onto the produced state information represented in p by the next state function δ cannot be distinguished, than this pair does not have to be distinguishable in the res set system. Below, the code for the $M_{S\to S}$ operator is shown. The code for the $M_{S\to O}$ is actually quite similar only now it not iterated over the pairs of states, but the compatible pairs of input symbols. $M_{S\to S}$ is quadratic in the number of states and linear in the number of input symbols, for $M_{I\to S}$ it is the other way around.

```
SetSystem& Machine::M_S_S( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol s = 0;  s < numStates()-1;  s++)
   for (symbol t = s+1;  t < numStates();  t++)
     {
       bool can_merge = TRUE;
       for (symbol i = 0;  can_merge && (i < numInputSymbols());  i++)
            can_merge = p.areCompatible( delta(s, i), delta(t, i) );
       if (can_merge) res->makeCompatible( s, t );
     }
  return *res;
}
```

```
SetSystem& Machine::M_I_S( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numInputSymbols() );
  for (symbol i = 0;  i < numInputSymbols()-1;  i++)
    for (symbol j = i+1;  j < numInputSymbols();  j++)
      {
        bool can_merge = TRUE;
        for (symbol s = 0;  can_merge && (s < numStates());  s++)
              can_merge = p.areCompatible( delta(s, i), delta(s, j) );
        if (can_merge) res->makeCompatible( i, j );
      }
  return *res;
}
```

Below, the code for the $M_{S \to O}$ and the $M_{I \to O}$ operator is shown. This code is also quite similar to that of $M_{S \to S}$ and $M_{I \to S}$ respectively. The main differences are that in case of a Moore machine, the result can be computed very fast. Secondly the mapping for these operators is done by the output function lambda (in stead of the next state function delta). $M_{S \to O}$ is quadratic in the number of states and linear in the number of input symbols, for $M_{I \to O}$ it is the other way around.

```
SetSystem& Machine::M_S_O( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol s = 0;  s < numStates()-1;  s++)
    for (symbol t = s+1;  t < numStates();  t++)
      {
        bool can_merge = TRUE;
        for (symbol i = 0;  can_merge && (i < numInputSymbols());  i++)
              can_merge = p.areCompatible( lambda(s, i), lambda(t, i) );
        if (can_merge) res->makeCompatible( s, t );
      }
  return *res;
}
```

```
SetSystem& Machine::M_I_O( const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numInputSymbols() );
  if ( isMoore() )
  {
    res->makeUnity();
    return *res;
  }
  for (symbol i = 0;  i < numInputSymbols()-1;  i++)
    for (symbol j = i+1;  j < numInputSymbols();  j++)
      {
        bool can_merge = TRUE;
        for (symbol s = 0;  can_merge && (s < numStates());  s++)
              can_merge = p.areCompatible( lambda(s, i), lambda(s, j) );
        if (can_merge) res->makeCompatible( i, j );
      }
  return *res;
}
```

For the $M_{I\times S\rightarrow S}$ and $M_{I\times S\rightarrow O}$ operators, set systems are defined on the Cartesian product of the input information and the state information, as well as, on the output information, respectively. So, there will be iterations over all pairs of compatible input and state / output information. Consequently, these operators are quadratic in the number of inputs as well as the number of states. The code for these operators is shown below.

```
SetSystem& Machine::M_SxI_S( const SetSystem& x, const SetSystem& p ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol s = 0;  s < numStates()-1;  s++)
   for (symbol t = s+1;  t < numStates();  t++)
     {
       bool can_merge = TRUE;
       for (symbol i = 0;  can_merge && (i < numInputSymbols());  i++)
         for (symbol j = 0;  can_merge && (j < numInputSymbols());  j++)
           if (x.areCompatible(i, j)) can_merge = p.areCompatible( delta(s, i), delta(t, j) );
       if (can_merge) res->makeCompatible( s, t );
     }
  return *res;
}
```

```
SetSystem& Machine::M_SxI_O( const SetSystem& x, const SetSystem& y ) const
{
  SetSystem *res = new SetSystem( numStates() );
  for (symbol s = 0;  s < numStates()-1;  s++)
   for (symbol t = s+1;  t < numStates();  t++)
     {
       bool can_merge = TRUE;
       for (symbol i = 0;  can_merge && (i < numInputSymbols());  i++)
         for (symbol j = 0;  can_merge && (j < numInputSymbols());  j++)
           if (x.areCompatible(i, j)) can_merge = y.areCompatible( lambda(s, i), lambda(t, j) );
       if (can_merge) res->makeCompatible( s, t );
     }
  return *res;
}
```

$M_{IB\rightarrow S}\{\varphi_S\}, M_{S\rightarrow OB}\{v\}$, $M_{IB\rightarrow OB}\{v\}$, $M_{I\rightarrow OB}\{v\}$ and $M_{IB\rightarrow O}\{\varphi_S\}$ are actually calculated the same way as described in the chapter on the theory as a combination of the ind (induced by input or output), the misp (minimal input support problem) and corresponding their corresponding M operators.

Some of the M and m operations are also implemented for partitions and block sets, however because they are not used in the currently developed last version of SeMaDe, they are not discussed in this report. For more information on the above discussed basic operators and algorithms implementations, the reader is referred to the source code of SeMaDe (machine.h, machine4.cxx, machine6.cxx).

3.3 Double Beam Searches

The double beam-search scheme was developed by Jóźwiak to robustly and efficiently construct some optimal or near-optimal solutions for synthesis problems [8]. In the decomposition method described in this report, it is used for the molecules sub-set selection and clustering. The double beam-search is a representative example of a deterministic parallel constructive search that can be applied to a broad class of problems.

To apply a constructive search for finding the most promising solutions to a certain synthesis problem, the problem has to be represented in terms of a space of states, where each state corresponds to a particular (partially constructed) solution, being a particular (partial) instantiation of the generic solution form. The **state space of (partial) solutions** is defined as an implicit tree, by means of:
– **initial state** *IS*
– the rules describing how to generate successors to each current state using **construction/move operators** *op* from a certain set *OP*, that realize transitions between states, and
– the **termination criteria** that (implicitly or explicitly) define a **set of goal states** *GS*, each representing a complete solution.

A particular **constructive search** searches (a part of the) state space, by starting from the initial state *IS*, and (selectively and/or in a certain order) applying the construction operators to the initial state or to the previously constructed states (in this way constructing the successive states), until some goal states from *GS* are reached.

The **basic beam-search** is a variation of breadth-first search, where only a limited number of the most promising alternatives are explored in parallel. It requires two **basic data structures**:
– **present states** *PS* (that contains the set of states which have been constructed earlier and are considered to be extended presently), and
– **candidate states** *CS* (that contains the set of states which are being created as a direct extension of the present states).
A third supplementary data structure, **final states** *FS*, contains the states that cannot be further extended.

The **basic beam-search** applies all possible to apply construction operators *op* from *OP* to each present state *ps* from *PS*, constructing this way a new generation of candidate states *CS*, and subsequently, applies its only selection mechanism **Select States** to the candidate states *CS* in order to include the best of them into a new generation of present states *PS*.

The **double-beam search**, developed by Jóźwiak as an extension and elaboration of the basic beam-search, uses **two selection mechanisms**: **SelectMoves** and **Select States**. Move operators are evaluated and selected in relation to a certain state *ps* (dynamically). Only a few of the most promising move operators are chosen for a certain present state *ps* by **Select Moves**, using problem-specific heuristic choice strategies and evaluation functions. By applying the selected move operators to each current state from *PS* a new generation of candidate states *CS* is created. The work of the second selection mechanism, **Select States**, is twofold: it scans the candidate states *CS* for states that cannot be further extended (represent complete solutions), in order to include them into the set of final states *FS*, and it examines the rest of the candidate states in order to include the best of them into a new generation of present states *PS*. The beam-search stops if the set *PS* becomes empty. Some of the constructed final states *FS* that satisfy some extra problem specific criteria are the goal states *GS*.

The selection mechanisms Select Moves and Select States must ensure that a solution that violates the hard constraints will not be constructed and they should try to robustly construct only some strictly optimal or near-optimal solutions by limited expenditure of computation time and memory space:
– Select Moves will select only those move operators that, applied to a given state, do not lead to the violation of hard constraints
– Select Moves and Select States will select only a limited number of the most promising operators or states, respectively, by using the estimations provided by some heuristic evaluation functions.

The selection mechanisms and evaluation functions determine together the search extent and quality of the results. In a number of constructive double-beam algorithms that where previously developed and tested, we implemented certain heuristic elaborations of the following **general decision rule** for selection of the construction operators and partial solutions: "***at each stage of the search take a decision that has the highest chance (or certainty) of leading to the optimal solution according to the estimations given by the evaluation criteria; if there are more decisions of comparable certainty, try a number of them in parallel, preferring the decisions that bring more information for deciding on the successive issues***".

According to the above rule, **Select Moves** will apply those move operators which maximize the choice certainty in a given present state and it will leave the operators which are open to doubts for future consideration. Since information contained in the partially constructed solutions and used by the evaluation functions grows with the progress of computations, the uncertainty related to both the partial solutions and construction operators decreases. In a particular computation stage, Select Moves will maximize the conditional probability that the application of a certain move operator to a certain partial solution leads to the optimal complete solution. Under this condition, it will maximize the growth of the information in the partial solution, which will be used to estimate the quality of choices in the successive computation steps, enhancing this way the choice certainty in the successive steps. The quality *Q(op)* of a given move operator *op* is decided by these two factors. Select Moves is controlled by two parameters:

– *MAXMOV ES*: the maximum number of the move operator alternatives explored in relation to each present state, and
– *OQFACT OR*: the operator quality factor.

In relation to a certain present state *ps* from PS, Select Moves selects no more than *MAXMOV ES* of the highest quality move operators *op* from OP, so that: $Q(op) \geq OQFACTOR * Qopmax$, where *Qopmax* is the quality of the best alternative operator that can be applied to a given state *ps*. Poor quality alternatives are not taken into account.

The task of **Select States**, in addition to selecting the final states, is to choose the most promising candidate states for a new generation of current states. Select States is controlled by two parameters:

– *MAXST ATES*: the maximum number of state alternatives explored at a certain computation stage, and
– *SQFACTOR*: the state quality factor.

Select States selects no more than *MAXSTATES* of the highest quality alternative states *cs* from *CS*, for which: $Q(cs) \geq SQFACTOR * Qmax$, where *Q(cs)* denotes the quality of an alternative *cs*

from *CS* and *Qmax* denotes the quality of the best state alternative in CS. Poor quality alternatives are not taken into account. *Q(cs)* can be computed by cumulating the qualities of the choices of move operators that took place during the construction of a certain *cs* and prediction of the quality of the best possible future operator choices on the way to the best complete solution possible to arrive at from this *cs*. Another possibility consists of predicting the quality of the best
complete solution that can be achieved from a certain candidate state *cs*. The double-beam search scheme is graphically represented in Figure 2. As signaled in Section 3, the selection of operators and partial solutions is generally performed with some uncertainty that decreases with the progress of computations, because both the "sure" information contained in partial solutions and the quality of prediction grow with this progress. Moreover, in the first phase of the search, the choices of operators can be made with much more certainty than the choices of partial solutions, because, in this phase, partial solutions almost do not exist and almost anything can happen to them on the way to achievable complete solutions. Therefore, in the first phase, the search should be performed almost exclusively based on the choices of operators and, with the progress of computations, more and more on the choices of partial solutions. In our double-beam algorithm, this is achieved by giving a relatively low value to *MAXMOV ES* compared to *MAXSTA TES* and a relatively high value to *OQFACT OR* compared to SQFACTOR. Since the uncertainty of estimations decreases with the progress of computations, *MAXMOVES* and *MAXSTATES* can decrease and *OQFACT OR* and *SQFACTOR* can increase with the progress of computations, increasing this way the search efficiency. In the first phase, the double beam-search is divergent to high degree, i.e. a large number of the most promising directions in the search space are tried. In the second phase, when it is already possible to estimate the search directions and operators with a relatively high degree of certainty, the search becomes more and more convergent. The highly divergent character of the double-beam search in the first phase, composed with the continuous and growing competition between the partial solutions in the second phase, result in its global character, effectiveness and efficiency. In this way, the double beam-search allows for an effective and efficient decision-making under changing uncertainty.

The double-beam search scheme was implemented in numerous circuit synthesis methods and tools solving different kinds of synthesis problems, and when tested on benchmarks, it efficiently produced very good results for all considered applications.

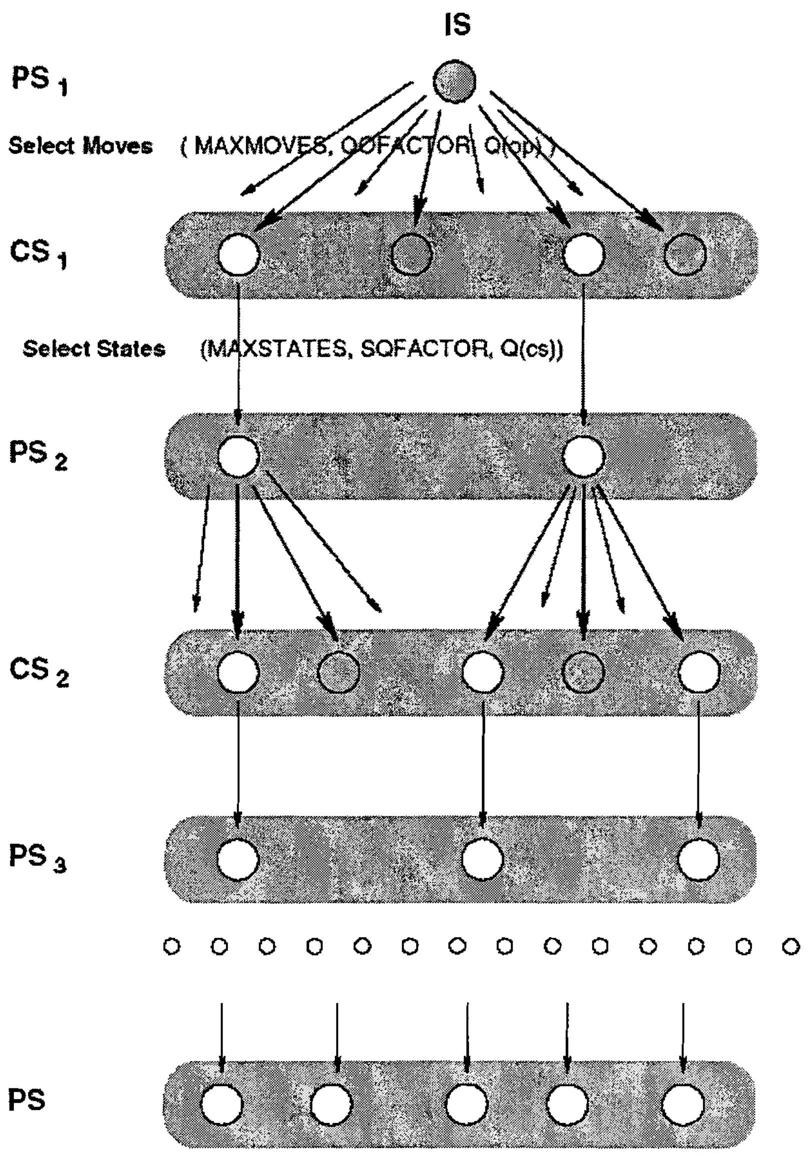

IS
PS 1
Select Moves (MAXMOVES, QOFACTOR, Q(op))
CS 1
Select States (MAXSTATES, SQFACTOR, Q(cs))
PS 2
CS 2
PS 3
PS

Figure 3.5: Overview of a beam search.

4. Main algorithms and their implementation

The aim of the information flow analysis is to collect and make explicit the information on the internal and external information flows of a given FSM that is relevant and will be used to control the correct-by-construction circuit generator delivered by the general decomposition theorem, to only synthesize the most promising circuit structure(s).

Using the molecule concepts and apparatus of Information Relations and Measures the internal information flows of the machine can be analyzed, made explicit and characterized. In this way, it is possible to see where and how certain information is produced and / or consumed and how difficult it is to produce some portions of information. By this analysis, coherent pieces of information can be found that can be easily produced / consumed together. By putting one or more of these coherent pieces of information together in a component machine, decomposition is found which well exploits the natural internal information flow structure of the original machine and will therefore result in a nice decomposition structure of relatively simple component machines, with only few interconnections between them and a limited number of long interconnections.

Besides internal information, also information on the external relationships of the sequential machine should be used to steer the decomposition towards a (near) optimal solution. By external information is meant e.g. information on where (physically) the used input / output pins are on the chip, or from / to what part of the data path certain signals are received / transmitted. By taking this information into account a different decomposition could be found then when only looking only at the internal structure (e.g. it could be better to produce / consume certain information near a part of the data path where many of these signals come from, or near some input / output pins. Also it could be better to produce some information more than once if the production is cheaper than de wiring otherwise required to transmit this information to its destination).

Using these two sorts of information (i.e. on the internal as well as external information flows) and confronting them with the actual synthesis objectives and constraints, heurist decisions will be taken to control the correct-by-construction circuit generator delivered by the general decomposition theorem, to only synthesize the most promising circuit structure(s).

We assume here that the external interconnections of an FSM being decomposed are decided before starting the decomposition, and consequently, the external information flows of the original FSM are fixed. In this situation, it is not difficult to characterize the external information flows, for instance, by specifying some clusters (sub-sets) of the related external interconnection signals and their characteristics. For example, one particular sub-set (cluster) of the FSM (controller) inputs can represent the status signals from a particular (part of) data path and another cluster of inputs the status signals from another (part of) data path. The geometrical closeness the corresponding (parts of) the data paths can represent the affinity between the input clusters and particular signals of the clusters. In a strictly similar way, the affinity between the FSM (controller) output signals (being control signals send to the particular (parts of) data paths can be modeled. Comparing to the internal information flow analysis of an FSM, the analysis of the FSM's external information flows is therefore not a difficult task. Moreover, it can only be performed knowing the topology of the system of data paths controlled by the given controller (FSM) and interconnections of the controller with the particular (parts of) data paths of the system. Therefore, we assume here that the external information flow analysis will be implemented in a tool that will call the decomposition tool, or alternatively, in a pre-processor to the actual decomposition or just a human designer, and the information on the external information flows will be delivered to the actual decomposition tool from the outside as one of the inputs by calling the decomposition tool. In consequence, only the FSM's internal information flow analysis will be considered in the remaining part of this section.

Through assuming that the external interconnections of an FSM being decomposed are fixed before starting the decomposition, we assumed in fact that a given FSM has assigned (binary) inputs and outputs. It has however just a single symbolic state variable, while all the partial FSMs in the final network of partial FSMs being the result from the decomposition will be assigned and only have the binary state variables. The decomposition into partial FSMs will thus perform a partial state assignment of the original FSM, and the particular partial FSMs will be further assigned using a networked version of an FSM state assignment program, as e.g. SEMADE. In such an FSM with assigned inputs and outputs and symbolic states, there are various information flows between the particular inputs and outputs (exclusively in the Mealy type FSM), inputs and the state variable, (current value of) the state variable and (next value of) the state variable, the (current value of the) state variable and the particular outputs. The information on each particular output is produced using particular state information, and in the Mealy case, particular input information from particular inputs. Each particular portion (e.g. atom) of the state information requires for its production particular state information and particular input information from particular inputs. From a particular sub-set of inputs only a certain maximal portion of state information can be produced that requires for its production also a certain minimum state information. The internal information flow analysis aims at discovering or making explicit the input-output, state-output, input-state, and (present)state-(next)state information relationships, especially from the viewpoint of the state information consumption and state information production. What portions of the state information are convenient from the consumption viewpoint (e.g. to produce information for certain outputs)? What portions of the state information are convenient from the production viewpoint (e.g. to be produced from certain small input supports and using only some small portions of another compatible state information)? What portions of state information are difficult to produce (e.g. require information from many inputs and/or many incompatible portions of state information)? What are the relationships between the different produced and consumed portions of state information and/or output information?

Based on the results of this analysis, the initial molecules will be created that represent some small partial machines of an initial decomposition that only serves to denote the initial results of the internal information flow analysis. Next, the initially created molecules will be analyzed and the relationships between the initially created molecules will be analyzed, and the molecule set will be improved, or in other words, the final molecules will be created that represent some small partial machines of an intermediate decomposition that are at the same time good both from the state information production and consumption, and output information consumption viewpoints.

4.1 Reading of the controller specification

First a description of the FSM that must be decomposed is read. This description consists of two parts contained in a text file. This file is in the .KISS format. The first part (which is always necessary) describes the controller's / FSM's behavior (it's original internal information flow). This file basically contains the transition table of an FSM with coded inputs, coded outputs and symbolic state variables. In this format any FSM can be described, also FSMs in other formats (e.g. VHDL, VERILOG, some graphical representation, etc.) can be converted to the .Kiss format by a one-to-one (direct) translation. The second part is optional and will be embedded in the .KISS file as comments started with a special character so that our tool recognizes it. This part will contain information on the external information flow (the surroundings of the controller/FSM) as described in the introduction of this report (also see figure 1). The precise format of this part is not determined yet, but it will contain some "groups" of inputs and "groups" of outputs of the FSM that are respectively coming from or going to a specific part of the data path (or are perhaps grouped for some other reason). Each of these "groups" also have a list of parameters (giving information on e.g. timing requirements, physical position, etc.) that apply to that particular group and/or particular inputs and outputs of that group.

4.2 *The Analysis Phase*

4.2.1 Creation of A molecules

After reading the .KISS file description of the sequential machine, the molecules A will first be constructed. The purpose of the molecules A is to produce enough output information to have a valid decomposition (see decomposition theory). In other words, the product of the output set systems of these molecules should be smaller than or the same as the set system induced by all outputs of the original machine: $\prod_i \beta_{Ai} \le ind\{U\}$.

We make an assumption that initially this covering of the output information is achieved by building a molecule A for every output bit of the original machine, where every molecule has to produce (at least) enough output information to compute this particular output bit: $\beta_{Ai} = ind\{u_i\}$ ($\beta_{Ai} \le ind\{u_i\}$) where $\bigcup u_i = U$. This is a good assumption to start the analysis process with, but later (after further analysis) during the improvement of the molecules or clustering, this assumption will not be made and it is very probable that a molecule may actually compute enough output information for more than one output bit, or only a part of the information needed for producing this bit.

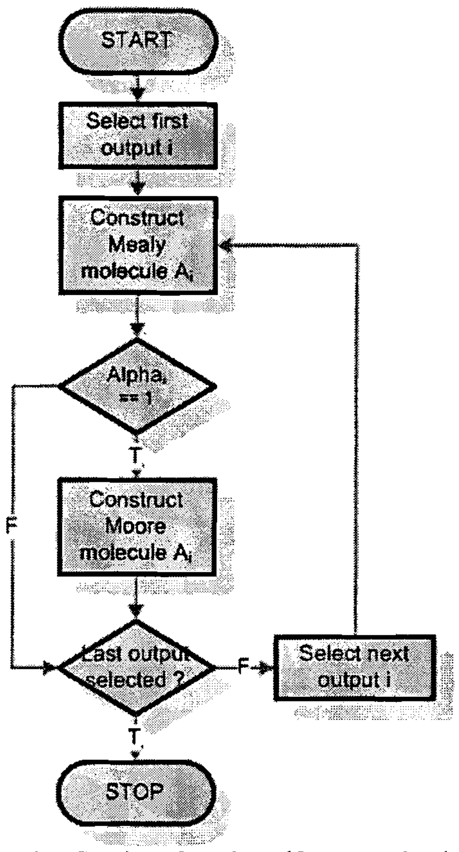


Figure 4.1: Creation of Mealy and Moore molecules A.

If during the construction of a molecule A for a particular output bit it will appear that no input information is necessary, a second (Moore) molecule will be constructed for this output. For this Moore molecule A, we will initially demand that all state information required for this particular output is computed locally.

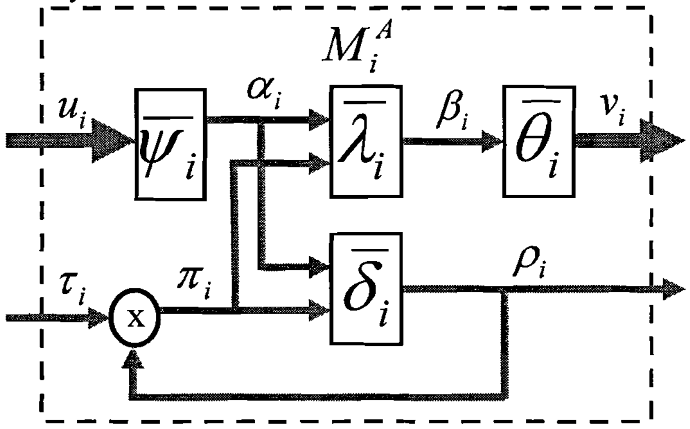


Figure 4.2: Internal structure of a Mealy molecule A.

As mentioned in the flow diagram, a molecule is constructed for every output. So the output support for molecule M_i^A is:

$$\mathrm{v}_i^{\min} = \mathrm{y}_i$$

The output support for this molecule is called $\mathrm{v}_i^{\min}$ because with the same input and state information required to produce this output, also other outputs can be computed. But the primary output bit y_i is the minimal output support that should be computed by this molecule.

The next step is to compute the set system induced by $\mathrm{v}_i^{\min}$:

$$\beta_i^{\max} = ind\left\{v_i^{\min}\right\}$$

$\beta_i^{\max}$ is the largest set system required to compute $v_i^{\min}$ (the minimal output support for this molecule). This translation between a set system and a support is represented by $\overline{\theta_i}$ in figure 4.2.

Now the input information that is needed to compute $\beta_i^{\max}$ is calculated, assuming all state information is present in this molecule:

$$\alpha_i^{\max} = M_{I\to O}\left\{\beta_i^{\max}\right\}$$

If $\alpha_i^{\max} = 1$, no input information is needed to compute the output, so this particular output is a Moore output. In this case, also a Moore molecule will be constructed for this output later on.

The input support required for this output is given by:

$$u_i = misp\left\{\alpha_i^{\max}\right\} = M_{IB\to O}\left\{\beta_i^{\max}\right\}$$

For this a minimal input support problem is solved with a quick-scan as well as a best first search algorithm is solved and the smallest support is used. This translation between a set system and a support is represented by $\overline{\psi_i}$ in figure 4.2.

All the input information that is present in this molecule, assuming no input encoder is used, is all the information induced by the input support:

$$\alpha_i = ind\left\{u_i\right\}$$

If however the input support u_i is too large, α_i is untouched, still allowing an input encoder because only the necessary information (to compute its output) is provided to the molecule.

During the computation of $\alpha_i^{\max}$ we assumed all state information was present in this molecule. In this step, the largest set system needed to compute $\beta_i^{\max}$ is computed, provided that input information α_i is present:

$$\pi_i = M_{S\times I\to O}\left\{\alpha_i, \beta_i^{\max}\right\}$$

Now that the imported input and state information is determined, all the state and output information that can be computed is determined:

$$\rho_i = m_{I\times S\to S}\left\{\alpha_i, \pi_i\right\}$$

And:

$$\beta_i = m_{I\times S\to O}\left\{\alpha_i, \pi_i\right\}$$

Finally the output support can be determined:

$v_i = Mosp\{\beta_i\}$

4.2.2 Creation of split A molecules

Now the Mealy molecules have been created and all output information is covered. The previous analysis however, is performed only from the viewpoint of the outputs (and their binary encoding). If the output information β_i of a particular A molecule does not contain any single portion of elementary information which requires all the consumed state information π_i and input information α_i to be supplied to this molecule, it is possible to split such a molecule according to its inputs (and the input's binary encoding).

From the creation of the Mealy A molecules the input support u_i of each molecule is known and the output information to be produced β_i too. By splitting the A molecules, the split A molecules should together still produce β_i however, each particular split molecule should have the smallest input support possible. Finding good input supports for these molecules is a combinatorial problem (similar to minimal input support). For an un-split A molecule where $|u_i| = n$, the number of supports containing k bits is determined by:

$$\binom{n}{k} = \frac{n!}{(n-k)!}$$

These sort of problems tend to explode exponentially (have extremely many solutions) for larger values of n and values of k around n/2. For a certain value n, the problem can be represented as an algebraic structure and the shape (geometry) of the problem is drawn in Figure 4.3.

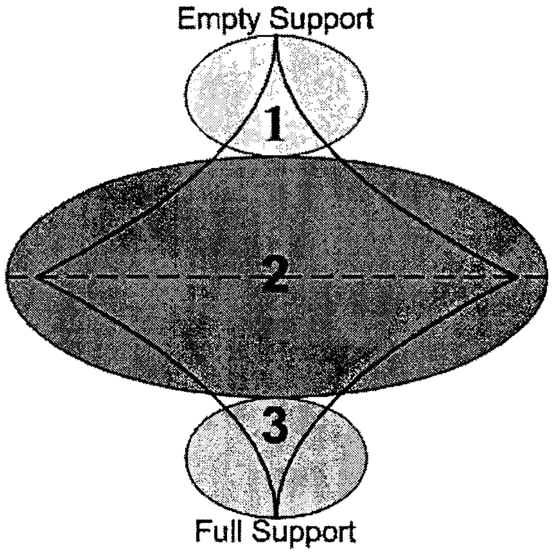


Figure 4.3: Geometry of a the combinatorial problem involving variable supports (lattice structures).

In this drawing k = 0 (Empty support) at the top and k = n (full support) at the bottom. Also the symmetry of this problem for k = n/2 is represented by the dashed line.

The problem has to be kept to a manageable size even for larger values of n. This is assumed by combining the following two computation reduction concepts:

1. A different kind of search is performed for the different regions of the problem indicated by 1, 2 and 3 in the figure.
2. The value for n is kept as low as possible. Also during the algorithm it is checked if n can be further reduced.

As noticed before, the problem can be divided into three regions. First, we discuss the properties of the molecules in each of these regions:

1. **Region 1** contains the molecules with small input supports. Also the solution space covered by this region is relatively small. In this region, we can perform an extensive search for good molecules very quickly. The molecules we find in this region will actually be the desired type because they can produce some necessary output information at a low cost related to the inputs required for their production. This information can also be allowed to be produced more than once since production of this information is relatively cheap compared to the cost of distributing this information to the partial machines that require this information.
2. **Region 2** contains the molecules with average size input supports, therefore they are not as desired as the molecules from region 1. Some output information however, may anyway require larger input supports then these of region 1. The main problem in this region is however, the fact that the solution space, related to this region, is very large. For this reason this space will be searched last (after region 1 and 3 are explored for good molecules). It will also be searched differently than regions 1 and 2: The starting point for these molecules will not be an input support, but some (elementary) output information that must be computed by a particular molecule. This will enable an efficient computation related to molecules concerning some elementary information not covered by the molecules from regions 1 and 3.
3. **Region 3** contains the molecules with a large input support and for this reason they are undesired. But also the difficult to compute information has to be computed, so there is no way to avoid the large molecules in general. Of course (with enough imported state information) these molecules can also produce all output information that is computed by all smaller supports. However, since (large) molecules in region 3 are undesirable, they will only be used to produce the essential output information (information that can not be produced by molecules with smaller input supports). We only allow this information to be produced once in the decomposition because its production costs are very high. Also during the molecule improvement, the molecules from this region are considered first, while still much freedom is left. Moreover, for these molecules the input encoders are considered to obtain a better distribution of the input information. Because the solution space covered by this region is also small, we find and analyze this hard to compute output information as soon as possible, because with this information covered we could try to reduce n (the number of input bits required to compute the state information), which will result in the corresponding reduction of the entire solution space.

The molecules in both regions 2 and 3 will be created with the possibility for input encoders $\alpha_i \neq ind(u_i)$. Moreover, in these regions a separate molecule will be initially created for a particular portion of elementary information (although some more information might be computed for free).

Summarizing: the direct search of the solution space in region 2 with method 1 or 3, e.g. through the input combination enumeration, requires to much time since there are too many supports in the middle region if n is large. Searching the entire solution space with method 2 would take to long since this would require an analysis of the machine on an atomic level. The combination of the three different specific methods for searching each of the three different regions results in an efficient search algorithm.

The second way mentioned to reduce the solution space was trying to reduce n. This is obtained by using the $M_{IB\to O}\{\beta_i\}$ operator on the remaining to be computed output information. With some output information already covered, perhaps one or more of the bits in the original input support are not needed anymore. Consequently, we find the minimum required input support (of size n).

As mentioned before, a molecule can be constructed given an input support u_i and a set system β_i which defines the output information that could be produced by a particular molecule. For a molecule with an input support in region 2, β_i is given, which is a portion of elementary state information. In contrast to methods 1 and 3, β_i has to be produced. In this case the input support u_i is determined during the construction of the molecule.

For method 1, u_i is given (by the algorithm discussed later). The input information:

$$\alpha_i = ind\{u_i\}$$

And the maximal amount of output information (smallest set system) that can be produced is:

$$\beta_i^{\min} = m_{I\to O}\{\alpha_i\} = m_{IB\to S}\{u_i\}$$

Also for method 1, not all information that can be produced has to be produced, because this is not necessary for a valid decomposition. Besides this, in general, not all information can be produced because the full input support is not present. So, the minimal set system of essential information that has to be computed by this molecule is:

$$\beta_i = \beta_i^{\min} + \beta_A$$

For method 3, the essential information is:

$$\beta_i = \beta_i^{\min} + \beta_A / \prod\nolimits_{|u_j|=|u_{i-1}|} \beta_j$$

So, only the information in β_A is computed which cannot be computed by molecules with a smaller input support.

Once β_i is determined, the state required state information and the output information that can be produced can be computed by:

$$\pi_i = M_{S\times I\to O}\{\alpha_i, \beta_i\}$$

$$\beta_i = m_{I\times S\to O}\{\alpha_i, \pi_i\}$$

$$v_i = Mosp\{\beta_i\}$$

For method 2 and, β_i is a portion of elementary state information that has to be computed by the molecule. In this case the input support is calculated by:

$$u_i = M_{IB\to O}\{\beta_i\}$$

The rest of the molecule is than constructed as follows:

$$\alpha_i = M_{I\to O}\{\beta_i\}$$
$$\pi_i = M_{S\times I\to O}\{\alpha_i, \beta_i\}$$
$$v_i = Mosp\{\beta_i\}$$

As it can be seen from the flowchart, methods 1 and 3 are alternated until some boundary is reached. After method 1 or 3 is complete (for a certain cardinality of inputs L_1 or L_3 respectively), β_A is updated. β_A is reduced with the information that was covered by the created molecules.

If $\beta_A = 1$ all the necessary state information is covered and the algorithm finished its computations for a particular molecule A (output). If not, the new levels are calculated and it is decided which method to use next. After method 1, method 2 or 3 can be called and after method 3, method 1 or 2 can be called. Method 2 is guaranteed to cover all remaining the state information and consequently, after calling method 2, the construction of molecules is certainly complete.

After method 1 has been completed at a certain level L_1 and some remaining β_A still has to be computed. The new level for method 3 can be computed:

$$L_3 = \min\left(L_3 - 1, \left|M_{IB\to O}\{\beta_A\}\right|\right)$$

Meaning that L_3 will be decreased at least by one if the problem could not be reduced.

Using method 2, for each portion of elementary information a molecule has to be created, so the number of molecules to cover all remaining state information is: $\left|IS(\beta_A)\right|$

Using method 3, $\begin{pmatrix} M_{IB\to O}\{\beta_A\} \\ L_3 \end{pmatrix}$ molecules have to be created at the next level (and perhaps some more if not all information in β_A was covered).

So method 3 is selected if:

$$\left|IS(\beta_A)\right| \geq \begin{pmatrix} M_{IB\to O}\{\beta_A\} \\ L_3 \end{pmatrix}$$

Otherwise method 2 will be selected.

The selection of method 2 or 1 after method 3 is done in a similar way.

The detailed procedure of switching between the different regions and is described below:

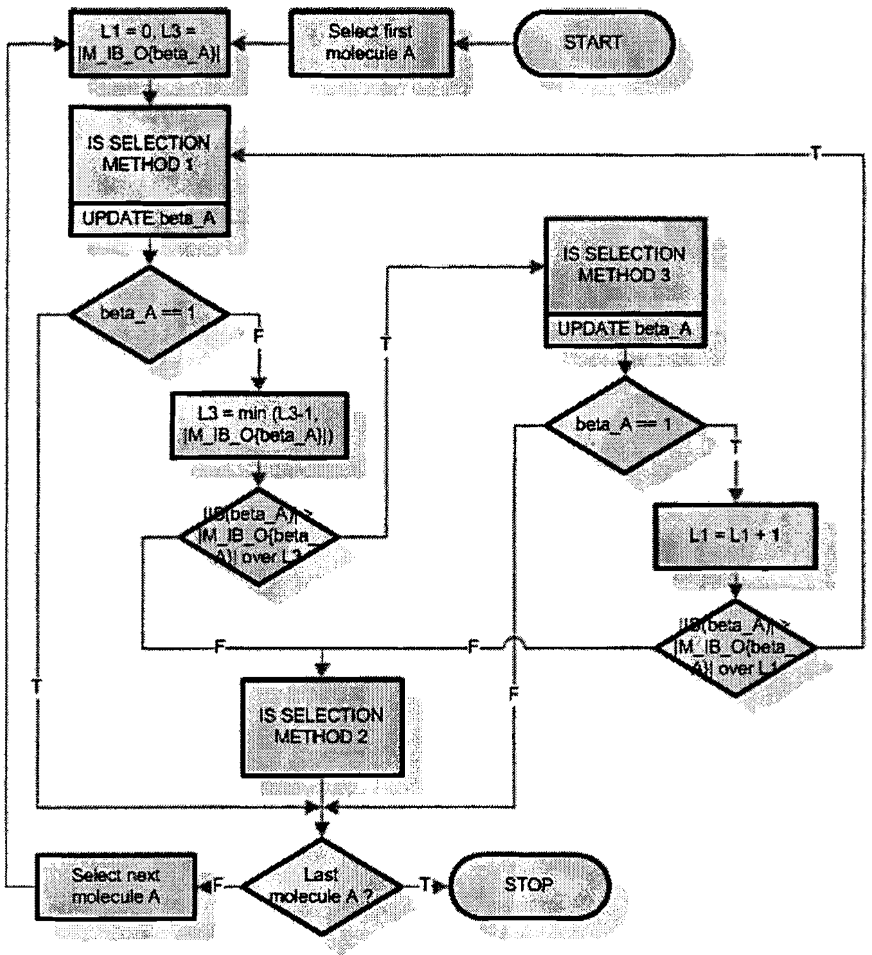


Figure 4.4: Flow chart for the splitting algorithm of A molecules.

4.2.3 Construction Moore molecules

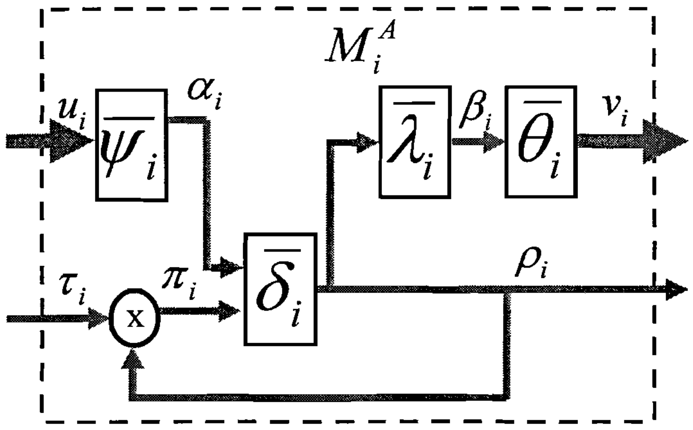


Figure 4.5: Internal structure of a Moore molecule A.

For a Moore molecule, to profit from it, all the state information necessary to compute a certain output must be computed locally. Since a Moore output doesn't require any input information, β_i is computed as follows:

$$\rho_i = M_{S\to OB}\{v_i\} = M_{S\to O}\{ind(v_i)\} = M_{S\to O}\{\beta_i\}$$

The state and input information necessary to compute this output are:

$$\alpha_i = M_{I\to S}\{\rho_i\}$$

And:

$$\pi_i = M_{S\times I\to S}\{\alpha_i, \rho_i\}$$

If the Moore molecule is suitable (which is decided later, during the molecule sub-set selection), an output molecule is constructed which has no output logic and has a two block set system for its state information.

4.2.4 Finding a maximal SP set system

For all partial machines and also all molecules it is allowed to import state information from other partial machines or molecules, respectively. For the entire decomposition this is not possible because it has to be a self-contained realization of the specified machine. So for the entire decomposition the substitution property for the state information must hold. Up to this point, the algorithm has performed the output analysis, which resulted in the creation of the A molecules (of the Mealy and Moore type).

The information required by all these A molecules is described by the following formula:

$$\pi_A = \prod_{i \in \text{molecules A}} \pi_i$$

So if π_A is available, all output information can be computed. This can also be achieved by computing all the state information required to compute $\pi_A \le M_{S\to OB}\{Y\}$.

The molecules A could already have computed some state information but because we want to analyze all the state information again from the viewpoint of the input information necessary for production of particular state information portions. We temporary assume that no state information is covered yet, so the globally computed state information is:

$$\rho_G = 1$$

So the information computed by the B molecules should be:

$$\rho_B \le \pi_A$$

However, to compute this state information, some more state information may be needed:

$$\pi_B \le M_{S\to S}\{\rho_B\}$$

In this way, $\rho_B = \rho_B \cdot \pi_B$ has to be and will be computed recursively until $\rho_B \le \pi_B$.

4.2.5 Creation of B molecules B

In chapters 4.4.1 to 4.4.3 the creation processes of the regular A and Moore molecules A were described. The molecules A make sure that all output information is computed. Moreover, some state information may already be computed for free here. In chapter 4.4.4 it was described how to find a maximal SP set system ρ_B in a recursive way. This ρ_B defines what state information has to be computed by all molecules B together. In particular, all the state and output information computed by the A molecules is will be covered again by the B molecules, but this time analyzed from the viewpoint of the input information consumption to compute it.

The next aim is to find a set of "good" molecules B that actually compute this information. By good it is meant here: low cost, e.g. that their input supports are as small as possible and they require only little state information (π_i should be large). The last is however of a smaller importance because we have much freedom to import state information (we can decide the encoding of the state information).

Similar to molecules A which are defined by an output support $v_i^{\min}$ or output information $\beta_i^{\max}$, molecules B are defined by a set system ρ_i (that represents the state information that we would like to be computed by that molecule) and/or an input support u_i. How a particular molecule B is computed from this ρ_i and/or u_i is described in further in this chapter. First, the algorithm for finding a good input support (u_i) and ρ_i for a molecule B is described.

As mentioned before, a molecule B can be constructed given an input support u_i and a set system ρ_B which defines the state information that could be produced by a particular molecule B. For a molecule B with an input support in region 2, ρ_i is given, which is a particular portion of elementary state information. In contrast to methods 1 and 3, from the requirement that ρ_i has to be produced, the input support u_i is determined during the construction of the molecule.

For methods 1 and 3, u_i is given (by the algorithm discussed later). The input information:

$$\alpha_i = ind\{u_i\}$$

And the maximal amount of the state information (the smallest set system) that can be produced is computed:

$$\rho_i^{\min} = m_{I\to S}\{\alpha_i\} = m_{IB\to S}\{u_i\}$$

For method 1, not all information that can be produced has actually to be produced, because this is not necessary for guaranteeing a valid decomposition. Also, in general, not all information can be produced, because the full input support is not present. So, the minimal set system representing the essential information that actually has to be computed by the molecule is the following:

$$\rho_i = \rho_i^{\min} + \rho_B$$

For method 3, the essential information is defined by:

$$\rho_i = \rho_i^{\min} + \rho_B / \prod\nolimits_{|u_j|=|u_{i-1}|} \rho_j$$

Here, only this information in ρ_B is computed which cannot be computed by molecules with smaller input supports and consequently, requires this large support of its computation. Once ρ_i

is determined, the state required state information and the output information that can be produced is computed by:

$$\pi_i = M_{S\times I\to S}\left\{\alpha_i, \rho_i\right\}$$

$$\beta_i = m_{I\times S\to O}\left\{\alpha_i, \pi_i\right\}$$

$$v_i = Mosp\left\{\beta_i\right\}$$

For method 2, ρ_i is a portion of elementary state information that has to be computed by the molecule. In this case the input support is calculated by:

$$u_i = M_{IB\to S}\left\{\rho_i\right\}$$

The rest of the molecule is then constructed as follows:

$$\alpha_i = ind\left\{u_i\right\}$$

$$\pi_i = M_{S\times I\to S}\left\{\alpha_i, \rho_i\right\}$$

$$\beta_i = m_{I\times S\to O}\left\{\alpha_i, \pi_i\right\}$$

$$v_i = Mosp\left\{\beta_i\right\}$$

The detailed procedure of switching between the different search regions 1, 2 and 3 is described below:

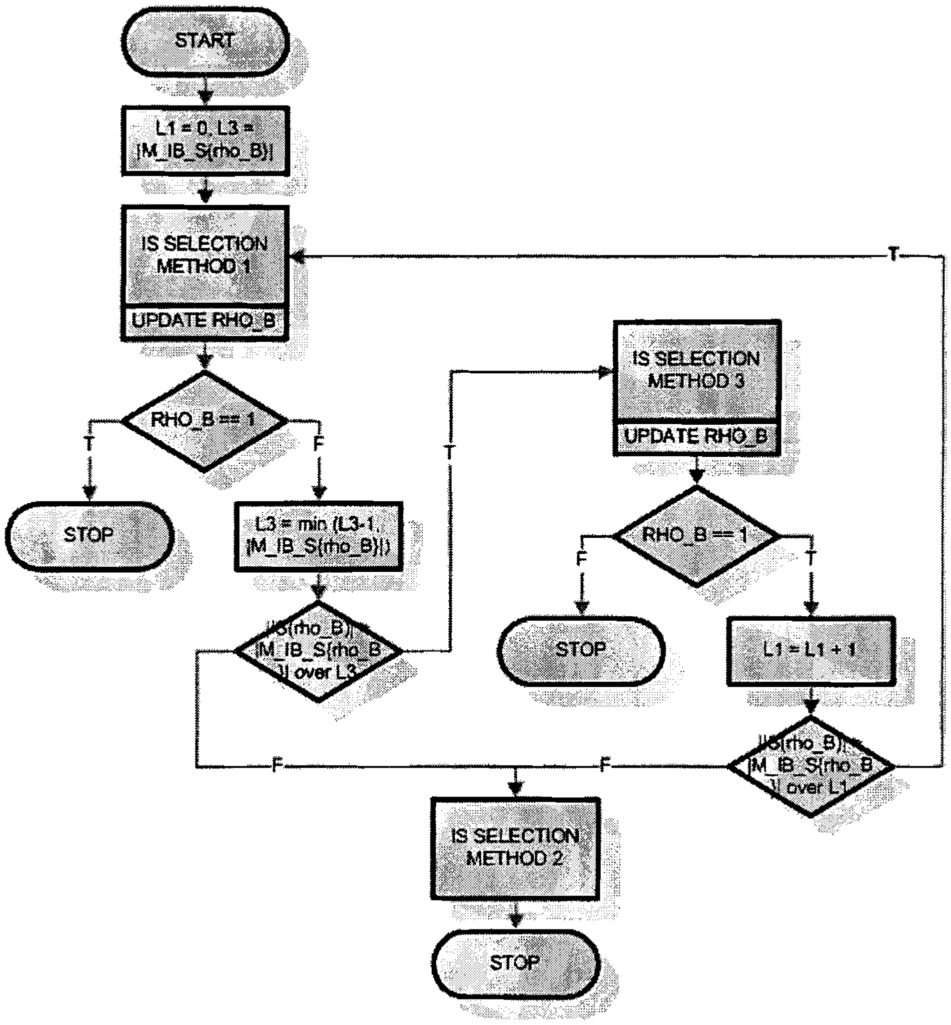


Figure 4.6: Flowchart for the creation algorithm of B molecules.

As it can be seen from the flowchart, methods 1 and 3 are alternated until some boundary regarding the input support cardinality is reached. After method 1 or 3 is complete (for a certain cardinality of inputs L_1 or L_3 respectively), ρ_B is updated. ρ_B is reduced with the information that was covered by the just created molecules.

If $\rho_B = 1$, all the necessary state information is covered and the algorithm finishes. If not, the successive levels are calculated and it is decided which method to use next. After method 1, method 2 or 3 can be called and after method 3, method 1 or 2 can be called. Method 2 is guaranteed to cover all the remaining state information, so that after calling method 2, the construction of B molecules is certainly complete.

After method 1 has been completed at a certain level L_1 and some remaining ρ_B still has to be computed. The new level for method 3 can be computed:

$$L_3 = \min\left(L_3 - 1, \left|M_{IB \to S}\left\{\rho_B\right\}\right|\right)$$

Meaning that L_3 will be decreased at least by one if the problem could not be reduced.
Using method 2, for each portion of elementary information a corresponding molecule has to be created. Thus, the number of molecules required to cover all the remaining state information is:

$$\left|IS\left(\rho_B\right)\right|$$

Using method 3, $\binom{M_{IB \to S}\left\{\rho_B\right\}}{L_3}$ molecules have to be created at the next level (and perhaps some more if not all information in ρ_B was covered).

Consequently, method 3 is selected if:

$$\left|IS\left(\rho_B\right)\right| \geq \binom{M_{IB \to S}\left\{\rho_B\right\}}{L_3}$$

Otherwise method 2 will be selected.

The selection of method 2 or 1 after method 3 is performed in a similar way.

4.2.6 Molecule improvement

In this phase the created molecules are improved. The main aim of this improvement is to consider not only the actual information flows for the molecules, but also the structure of this information. This is an iterative process which means that the improvement continues until no further improvement is possible. During this improvement phase, the relations between the created molecules are examined. During this analysis a number of alternative ways for improvement are considered:

- **Redundancy elimination**: Here is looked at the possibility of producing less information in a molecule (A) than required for computing its corresponding output bit. This **can** only be realized if the other information (not produced anymore by the molecule itself) is produced by the other molecules. The idea behind this is that the information from several molecules necessary to compute a particular output variable, information can be combined during the clustering phase (by an output decoder or by the clustering itself). Redundancy elimination **should** only be performed if there is enough gain (by reducing the input and / or state information, input support and the number and size of the input encoders required).

- **Molecule splitting**: This is analogues to the redundancy elimination, in the sense that a molecule is not required to produce all the information necessary to compute a particular output bit. The difference is, that the output information that is not produced by the molecule itself does not have to be covered by the other molecules. Instead a number of molecules will be created from the original A molecule that together produce enough output information to calculate the corresponding output variable. In this way, the molecules necessary for computing the output bits that require large input supports and / or state and input information can be replaced by a number of smaller molecules. This splitting is desired before the clustering phase, because during the clustering phase, molecules can only be merged. They may only get larger and consequently, later there will be no chance of their improvement by making them smaller.
- **Molecule joining**: Here it is analyzed for a molecule if the molecule will be able to compute more than one output bit by adding no (or only little extra) input/state information. In this way, some other molecules covering the computation of the additional outputs can be removed (they may also be preserved as a possible alternative to computing a certain bit). Molecule joining is not to be confused with molecule merging. Molecule merging is performed during the clustering phase of the method and combines molecules that have high affinity.

As mentioned earlier, the improvement phase is an iterative process, so the mechanisms as described above can be repeated a number of times (e.g. a molecule can be joint and than be split again). In this way, the computation of the required state and output information can be divided over the different molecules in a desired way, and possibly some earlier incorrect decisions based on less information can be corrected when more information is present to support the decision making. This process continues until there are no changes anymore, or if some other stop condition is met.

The same mechanisms (redundancy elimination, splitting and joining) can also be applied to the input encoders. These improvements will be made (every iteration) after the molecules have been improved.

The "Moore" molecules A, corresponding to the outputs directly used as state variables for the state assignment, cannot be improved (much) anymore. This is because all the state and input information has to be present in one place to be able to compute the particular Moore output and its corresponding state variable. Thus, the molecule cannot be decomposed further. It is also not useful to add much input or state information because of the fact that the ρ set system of the molecule may only have two blocks (which are mapped from the state information to output information so that the corresponding output logic is reduced to a single wire). The only thing that could be done is to reduce the number of don't cares (if there are any) in the ρ set system in the best way. These don't cares are symbols which are present in both the blocks of the ρ set system. In this way the ρ set system can contain more information, but this requires more input and consumed state information to be sent to the molecule. In most cases this information will not be compatible and cannot be fully exploited (only the information which reduces don't cares can actually be produced in this molecule). So improvement of these Moore molecules A is not significant and will be postponed or not done at all. Only after all improvements are complete, there has to be decided which Moore molecules are used.

For the "Mealy" molecules A there are more possibilities for improvement because the restriction that the ρ set system may only have two blocks doesn't hold. Also the Mealy molecules A may be split further according to their input supports in a way similar to the creation of the B molecules. The main difference is that now all output information for a particular output variable should be covered (instead of all state information).

Further improvements are focused on the split mealy molecules A and the B molecules. First the B molecules are sorted according to the size of their input support and secondly (in case of the same input support) according to the cardinality of the pi set system. In this way, the molecules that compute state (or output information) that requires a lot of inputs are considered first, while still much freedom is left.

For each molecule, the ρ_i (or β_i beta in case of mealy molecules A) set system will be examined. This set system contains the essential produced state information for the corresponding molecule, meaning the information that cannot be computed with a molecule with a smaller input support.

First of all, the number of essential blocks will be determined. An essential block is a block that contains one or more symbols that are not present in any other blocks. This number of essential blocks is a lower bound for the number of blocks needed to represent the essential state information produced by this molecule. Another way to perform this is by coloring of all the blocks in:

$$m_{IB\to S}\{u_i\} \text{ or } m_{I\to S}\{\alpha_i^{\max}\}$$

for examining improvement without or with an input encoder respectively. This coloring gives the minimal number of blocks (colours) by merging blocks while no essential information is lost. This number will be round up to the nearest power of 2 to maximally exploit the interconnections between the molecules.

Different numbers of blocks in set systems (differently large molecules) will be allowed, depending on the size of the original machine. Only 2 or 4 block set systems will be allowed for average size machines. For smaller machines, only 2 block set systems are allowed. Exclusively for the extremely large machines, set systems with ore blocks may be allowed. This is due to the fact that during the clustering molecules will only be merged and as a consequence, the number of state variables for each particular sub-machine after clustering would become too large for set systems with more blocks. If the number of blocks is too large, for some molecules, the molecules can be split by constructing an input encoder which supplies the input information more efficiently to the molecules. For smaller molecules, the cost of having more molecules with the same support is not too high and so several molecules with the same small or similar support, each computing a part of the essential information, can be constructed.

Once all molecules are small enough regarding their input supports and produced state information, the produced state information should be finally divided over the molecules in a way that information that is hard to produce (requires much state and / or input information) is produced only once, in one place and is then transported to all the places where it is needed. And all the information which can be easily produced may be produced everywhere, where it is needed for some computations and perhaps also in some other places where it is only needed for increasing the compatibility of the set systems (less blocks). Of course, the cases between thos too extremes are also possible.

To find good candidates for the set systems, first of all the essential information should be produced. Any freedom left at this point can be used to improve the amount of information which can be transferred on a certain amount of bits. If there is still a large amount of freedom left, not all possible set systems can be constructed and examined, but we need heuristics which will give us an indication for promising set systems. First, we have the local analysis. This analysis can be done only focusing on particular molecule. This analysis only tries to find set systems that have as much information as possible while requiring only a small amount of state information. However, not only the amount of information per bit is important, but also the way how it is consumed and the way in which other molecules produce information is important. This is why also a Global analysis has to be done. This global analysis is more important, but also more time consuming, so the actual order in which they are performed and the weight assigned to their importance has to be established by experiments. However, if different heuristics suggest different set systems, they can be considered in parallel as alternative solutions.

Local analysis heuristics:

1. The number of blocks for the ρ_i and α_i set systems should be as small as possible (2 or 4 blocks).
2. Blocks should approximately have equal size.
3. A particular symbol should be in as few blocks as possible (this results in few don't cares)
4. $\left|M_{S\times I\to S}\{\alpha_i,\rho_i\}\right|$ should be small.

Items 1 to 3 make sure that the information per bit is maximized. And item 4 makes sure that not too much state information is needed to produce a given portion of state information. Here, the amount of information is of more importance, because is has to be covered by ρ_i set systems, and since these all have as much information per bit as possible, the number of bits needed to cover the imported state information will be close to minimal.

Global analysis heuristics:

1. While merging blocks of the set system representing the produced state information, trying to keep as much information from larger molecules. In this way molecules can be reduced (or can be removed entirely), because they do not have to produce some (or all) essential information any more.
2. Construct set systems which will have as much blocks as possible of approximately equal size when multiplied with constructed set systems from other molecules. If this condition holds the set systems will be almost orthogonal, which is an indication of a good (close to) minimal state assignment.
3. Try to construct set systems which resemble the information as it is consumed by certain molecules.

Using these heuristics a number of molecules are constructed. If (almost) all heuristics suggest a certain molecule, only this molecule has to be constructed. If however heuristics contradict more alternatives have to be examined and only when all good alternatives have been considered, the best decomposition(s) can be determined.

4.2.7 Molecule sub-set selection

Since, during the improvement phase multiple alternatives where considered to produce some portions of state or output information some redundancy was introduced. The final task of the analysis part of the method is the initial creation of (a) valid decomposition(s). This is done by selecting one or more most promising sub-sets from the pool of all molecules created so far. Each of these sub-sets represent in fact an early prototype of a valid decomposition that will be further processed through the clustering and FSM network construction phases, to finally result in one or more actual promising decompositions.

The first hard constraint posed on these subsets is that the decomposition represented by the sub-set is valid. During the construction and improvement it is guaranteed that for all the molecules (at the local level), the constraints of the decomposition theory are satisfied. However for each sub-set also the global constraints have to hold (enough output and state information has to be produced, respectively):

$$\beta_S = \prod_{i \in S} \beta_i \le ind(Y) \text{ and } \rho_S = \prod_{i \in S} \rho_i \le \pi_S = \prod_{i \in S} \pi_i$$

These sub-sets (or prototype solutions) can be seen as supports, where a full support contains all molecules (which is always valid). And an empty support (which contains no molecules) is always invalid. If a solution with only one molecule is valid, this is a trivial case of a decomposition into one submachine. We will be only interested in the remaining natural and valid cases.

This problem, of finding the most promising valid molecules sub-sets, is equivalent to set covering, which resembles the row covering / minimal input support problem, with the main difference that the cost function is not equal for all molecules. This set covering problem however, does not have to be solved for all created molecules so far, but can be split in a number of sub-problems. This is possible, because we have some additional information on how the molecules were created. First, we will explain how this subset selection works for Moore machines. Secondly a suggestion is made on how to extend this algorithm for the Mealy machines.

Some sequential machines may contain one or more outputs which only require state information to compute this particular (Moore-type) output. In a Moore machine this is true for all outputs, but also some Mealy machines may contain one or more of such outputs. As discussed earlier, for these outputs special Moore molecules were created. These molecules are very useful because they do not only compute an output, but also induce a two block set system on the states that can be directly used to represent a binary state variable. Secondly, if actually used in a decomposition, they require no output logic (only a wire from the corresponding state variable to the output variable).

For these reasons our algorithm for creating valid subsets of molecules starts by selecting a subset of these Moore molecules. This subset does not have to be valid by itself because it can be made valid by adding B molecules to produce the remaining state information.

Our initial guess is to use all constructed Moore molecules. The level parameter is set to the number of Moore molecules created, in the rest of this algorithm this level will indicate the number of Moore molecules that will be in the Moore subset of the newly created subsets.

After the final Moore subset is determined, the prototype decomposition has to be made valid. As mentioned earlier, this is guaranteed by adding some B molecules. Once the decomposition has been made valid, the cost of this decomposition has to be calculated to determine the quality of this particular solution/decomposition. This process is called **complete**, and is explained in more detail later in this chapter.

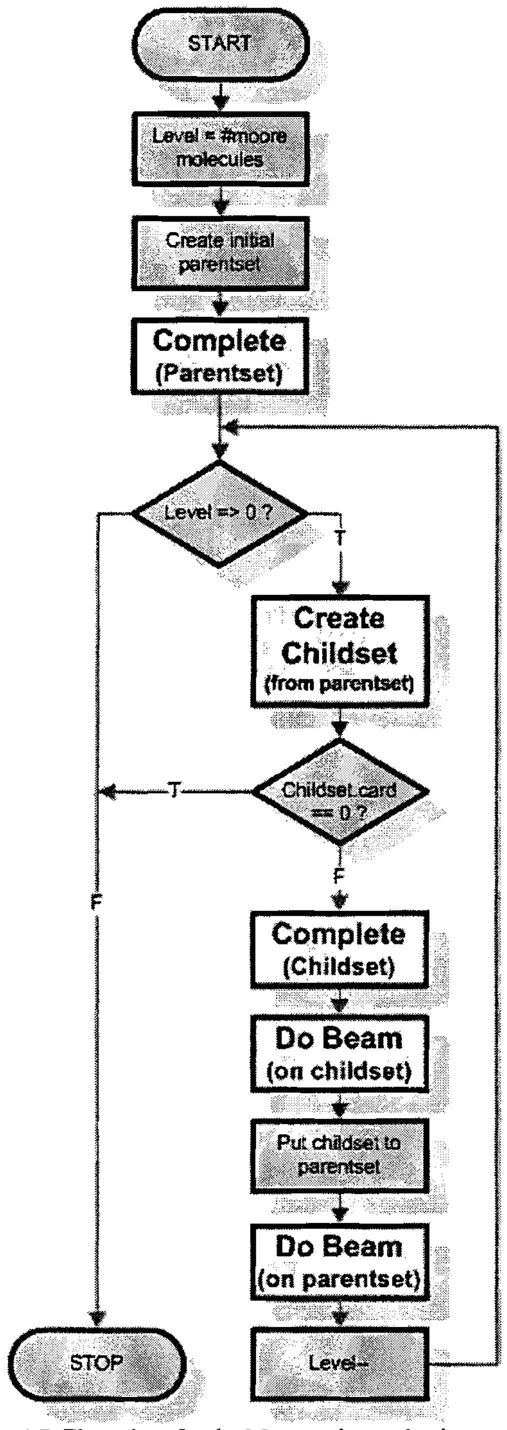


Figure 4.7: Flow chart for the Moore subset selection process.

At the start of the Moore molecules sub-set selection process, we have an initial parent set which contains just one solution. We check if the level (number of molecules) is larger or equal to 0. In the case of a Mealy machine with no Moore outputs it will be 0 from the start and all the state information will be produced by the B molecules selected by the complete process. In the case of only one Moore molecule, the choice is simply whether to use this Moore molecule or not. In all the remaining cases, the following more complex selection process will be performed.

First, the children are created. For each solution in the parent set at the current level (where the number of Moore molecules in their subset is equal to level). The children of a particular parent are created by removing one Moore molecule from the parent set at a time. Consider please this following example. At a certain point in the algorithm, namely level 5, the parent set contains Moore subsets: {0, 1, 2, 3, 4, 5}, {0, 2, 3, 4, 5}, {0, 1, 2, 4, 5}, {0, 1, 2, 3, 4}. Note that {1, 2, 3, 4, 5}, {0, 1, 3, 4, 5}, {0, 1, 2, 3, 5} where constructed at the previous level but where rejected by the first or second beam as will be discussed later in this chapter. At this level {0, 1, 2, 3, 4, 5} is not used to create children because at the previous level it's children where already considered. {0, 2, 3, 4, 5} will have children {2, 3, 4, 5}, {0, 3, 4, 5}, {0, 2, 4, 5}, {0, 2, 3, 5}, {0, 2, 3, 4}. {0, 1, 2, 4, 5} will have children {1, 2, 4, 5}, {0, 1, 4, 5}, {0, 1, 2, 5}, {0, 1, 2, 4}. Note that {0, 2, 4, 5} was also a child of {0, 2, 3, 4, 5} so it will not be created again. Finally {0, 1, 2, 3, 4} will have children {1, 2, 3, 4}, {0, 1, 3, 4}, {0, 1, 2, 3}. Again {0, 2, 3, 4},{0, 1, 2, 4} are not created because they where already present. So the three solutions of the current level produced 5+4+3= 12 children. Note that this is already a subset of all possible sub-sets of 4 Moore molecules (which has 6*5*4/3*2*1 = 120/6=20 children). This is due to the fact that at the previous level (in this case only level 5, not all subsets where accepted by the double beam).

If no children where created because there where no parents (no solutions with "level" Moore molecules), the newly created child solutions are completed, this completing process is the same as the completing of the initial parent set. After they are completed, only the best of the created children will be selected for the new generation of solutions and they can be present as final solutions. This is done by the first beam, which operated only on the child set. Subsequently, the children are added to the parent set and now the second beam is performed. The second and the first beam are exactly the same except for the fact that the second beam operates on the parent set (not only the children), which at this point contains all solutions created and accepted so far. Finally, the level is decreased by one and the loop is entered again as long as the level is larger or equal than 0.

The **beam operators** will be discussed in somewhat more detail. As mentioned earlier, the only difference between the two beams is the set that they operate on, so the beam algorithms is actually implemented only once. The beam has two parameters: Q, the quality factor and P, the maximal number of solutions that are allowed. First the best solution (with the lowest cost): minimal cost is found, and the quality threshold is determined Qth = minimal cost * Q (where Q larger of equal to 1). All solutions with a lower cost that Qth are accepted. If the number of accepted solutions is larger that P, the accepted solutions are sorted and only the best P solutions are accepted.

Finally the **completing process** will be described in somewhat more detail. At the point where completion is called, the subset of Moore molecules is already known. This means that:

$\pi_{Moore} = \prod_{\text{i is in Moore subset}} \pi_i$ and $\rho_{Moore} = \prod_{\text{i is in Moore subset}} \rho_i$ are already known.

Also $\text{sinfo} = M_{S \to O}\{ind(Y)\}$ which is the information needed to compute all outputs is known.

The aim of the complete process is to find a sub-set of B molecules such that:

$\pi_B = \prod_{\text{i is in B subset}} \pi_i$ and $\rho_B = \prod_{\text{i is in B subset}} \rho_i$ are fulfilling the following two requirements:

$\rho_{Moore} \cdot \rho_B \leq \pi_{Moore} \cdot \pi_B$ and $\rho_{Moore} \cdot \rho_B \leq \text{sinfo}$.

For finding the B sub-set two methods are implemented. Similar to the way the Moore subsets are determined. We start with all B molecules, remove one B molecule at a time and determine the solution cost. If $\rho_{Moore} \cdot \rho_B \leq \pi_{Moore} \cdot \pi_B$ and $\rho_{Moore} \cdot \rho_B \leq \text{sinfo}$ are met. The first and the second beam accept only the best solutions of the newly created and all created subsets of B molecules so far, respectively. Once this is done, the single best subset is chosen from the possibly many final subsets that result from the search.

This first method described here gives good results, but at this stage the cost function is not yet accurate (because clustering is not yet performed). Also since the B molecules have similar costs, the width of the beam is mostly determined by quantity instead of quality. Because of this poor distinction of the quality of the solutions the search takes quite a long time. This is why the second method was developed.

The second method is based on a recursive QuickScan. Initially, before the first iteration $\pi_B = \rho_B = 1$ (no B molecules are selected). If $\rho_{Moore} \cdot \rho_B \leq \pi_{Moore} \cdot \pi_B$ and $\rho_{Moore} \cdot \rho_B \leq \text{sinfo}$ are met, the algorithm is finished. If not the information that needs to be covered is: $\pi_B \cdot \pi_{Moore} \cdot \text{sinfo}/\rho_{Moore}$. Now QuickScan is used to select B molecules to cover the required information for the first time so π_B and ρ_B can be updated. Then again the conditions $\rho_{Moore} \cdot \rho_B \leq \pi_{Moore} \cdot \pi_B$ and $\rho_{Moore} \cdot \rho_B \leq \text{sinfo}$ are checked. This loop continues until these conditions are met.

The main advantage of the second method is speed, and that QuickScan gives one result. Since only one result is accepted, no search has to be performed for the best one. The main disadvantage of this method is that the difference in the costs of the molecules is not considered. However because of the way the B molecules are created, they are all of similar sizes and costs, so, this poses no real problem. Secondly, most of these molecules produce essential information, so if this information is needed they should be added no matter the cost. Finally, clustering still has to be performed, the cost (especially for the B molecules) is not really accurate at this point. These reasons make the QuickScan approach a good and faster alternative to method 1.

Once the Moore, as well as the B sub-sets are known, the complete process can also calculate the total solution cost, which is the sum of the costs of all the clusters in the Moore subset and the clusters in the B subset. How the cost of a single cluster is calculated is explained in Chapter 4.6 because the same cost measure is also used to determine the quality of the second beam of the clustering process.

This process is easily extended, to allow for the molecules sub-set selections for the Mealy machines also. The main difference is that the Moore subset selection algorithm described above (which creates a subset of Moore type A molecules and B molecules) should be preceded by an algorithm which selects molecules from the set of A and split A molecules that produce the output information to create a valid decomposition. This algorithm involves two main questions to be solved. The first question is, what information should actually be covered and the second is how then to cover this information.

A starting point for the first question is that at least the computation of all output information has to be covered: $\beta_G = ind(Y)$, where β_G is the global output information and Y is the full output support of the machine.

Of coarse, some of this information can be produced from the state information only, without input information (e.g. Moore outputs). So $\beta_G = {}^{ind(Y)}\!\Big/_{M_{S\times I\to O}\{1, ind(Y)\}}$. This constraint is actually much simpler than that for the state information because there is no recursion involved. Once the set system β_G is determined, only a subset of A molecules has to be found that satisfies this constraint.

First, we find if there where any A molecules that could not be split further (because they produce some output information that requires all the input and state information also required for computation of the entire output). This (elementary) portion of output information that makes the molecule A unsuitable for splitting is always essential (because it is produced nowhere else). So these un-split molecules can be added to the flagged set of every solution without further analysis. The remaining information may be reduced by all output information that is produced by these molecules because this information is computed by the already selected molecules.

Secondly, if not all output information in β_G is covered yet, we find which of the remaining to be covered output information is essential (produced only by one split molecule A). These molecules also have to be present in all solutions. However, at this time they are added to the non-flagged set, because they may require further clustering.

Finally, if some output information is still not covered, we can use a method similar to the selection of B and Moore molecules to cover the remaining output information. E.g. add all remaining split and full A molecules to the initial set, and remove them one at a time al long as it holds that the molecules selected so far produce at least β_G. This is for the first time in this algorithm that multiple solutions could be required, because the already selected essential molecules are completed by the subsets of remaining molecules with the lowest cost.

4.3 The Clustering Phase

4.3.1 Aim of clustering

The aim of the clustering process is clear in the context of the work of the whole decomposition process an din the context of the FSM information flow analysis part. All major aspects of an adequate information structuring from the viewpoint of the state information production and consumption and output information production are taken into account during the molecule creation phase, molecule improvement and selection of the most promising sub-sets except that the portions of the state information in some of the molecules may be too small, and in consequence too many, too small partial machines would result from it. Therefore, ***in the clustering phase the main aim is to cluster together some relative molecules, especially such that at least one of them contains a too small portion of state information, or if two molecules are very similar to each other.***
The relation of molecules is represented by their affinity that is defined through a system of affinity measures described in Section 4 of this clustering method description.

4.3.2 Input data

The input data to the clustering algorithm is the output acquired from the analysis part. One particular valid decomposition generated by the analysis algorithms is called a solution. A solution is a valid prototype decomposition, meaning that enough state and output information is produced. A solution is basically a set of clusters (or molecules) that are divided in two different sub-sets, the **flagged-** and the **non-flagged** set. The **flagged set** contains the molecules of which is already clear during the analysis phase that no changes should be made to them and that each of them should be present in the final decomposition as a separate partial machine. Therefore, the clustering process will not consider the molecules in this set for clustering. The **non-flagged** set however contains the molecules that may be still too small to result in a good partial machine or have a very high affinity and should be considered for further clustering. Thus, so the clustering algorithm will operate only on this **non-flagged** set of molecules.

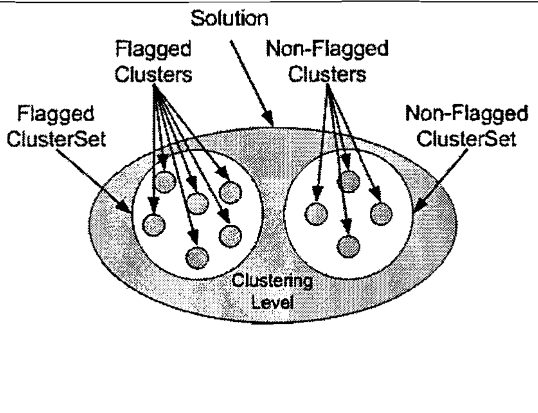


Figure 4.8: Overview of Solution.

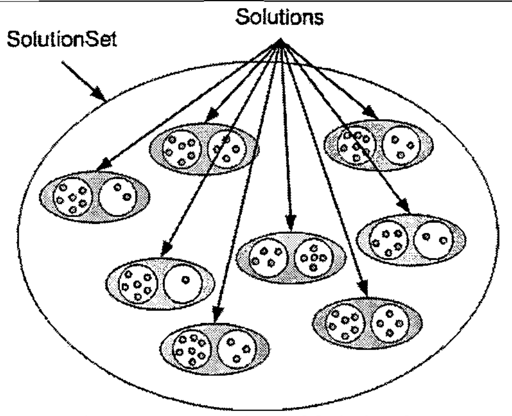


Figure 4.9: Overview of Solution Set.

However, a particular solution represents only one option that leads to one or more final decomposition(s). During the analysis phase, it may not be the case that one option is found which is clearly superior to all other options. Often several most promising solutions are of comparable quality and this quality is estimated using a heuristic measure that only has a strong positive correlation with the actual circuit quality. To cope with this uncertainty, the analysis phase will in general construct more than one promising solution. These solutions are stored in a solution set as depictured in Figure 4.9.

4.3.3 The main Clustering Algorithm

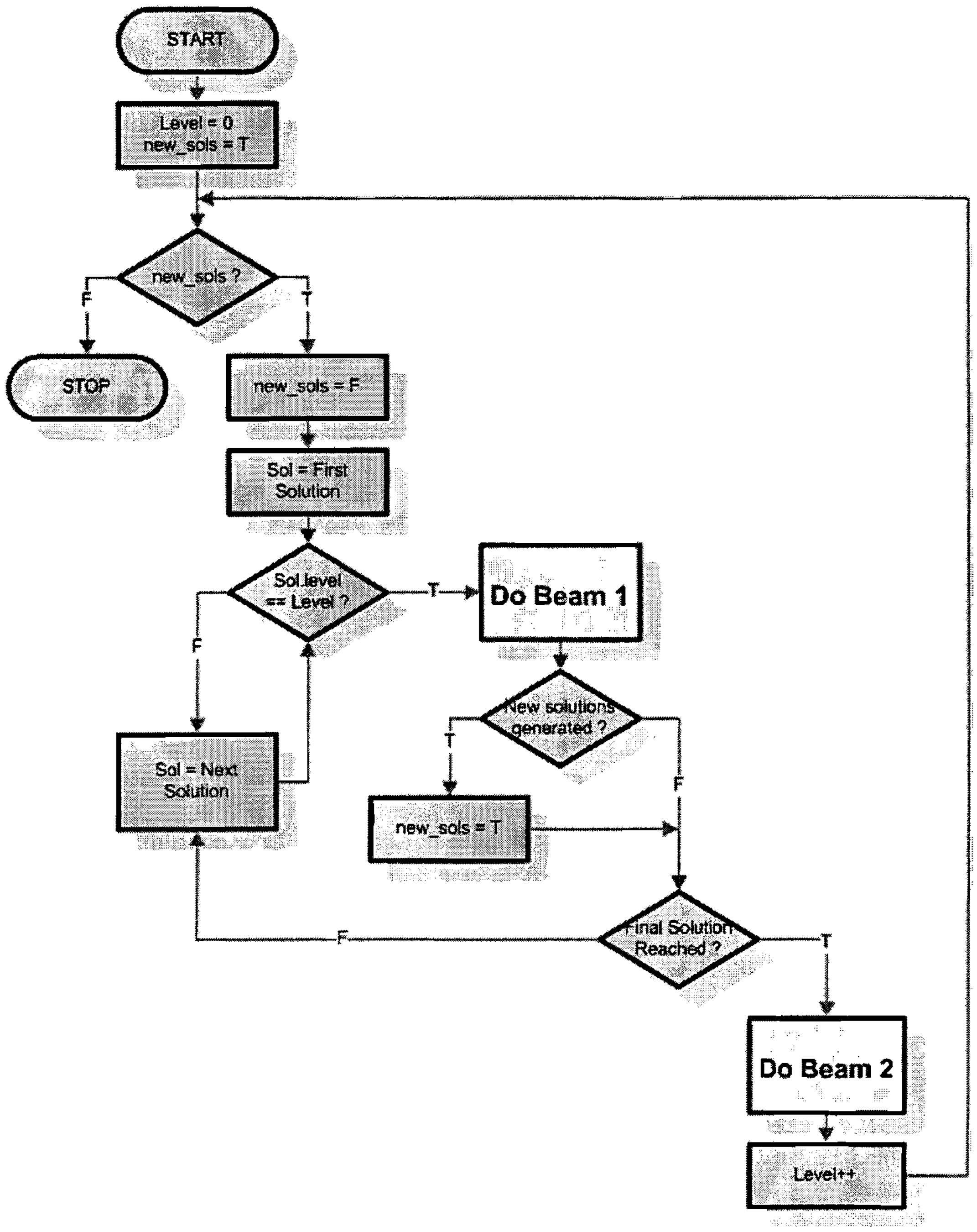


Figure 4.10: The main clustering algorithm flowchart.

The clustering algorithm is based on the double beam search scheme [8]. This is visible in the flowchart by the "Do Beam 1" and "Do Beam 2" blocks. These two blocks are described in more detail in chapters 4.5 and 4.6 respectively. In this chapter the purpose of these two beam processes is described and also the stop condition for the clustering process is explained (that is clearly visible from the main flowchart). To further illustrate the main algorithm Figure 4 gives an overview of a possible clustering.

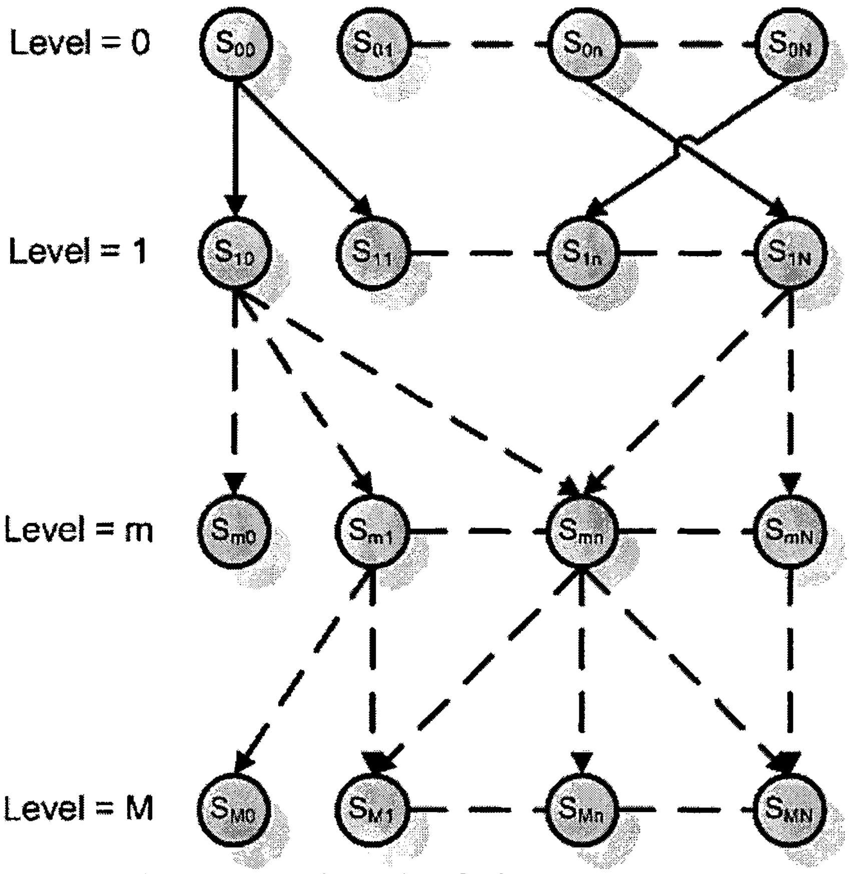


Figure 4.11: Example overview of a clustering process.

From the analysis phase, one or more solutions are passed on to the clustering phase in the solution set. All these initial solutions have first index zero, which is the clustering level. The second index is an identifier for a particular solution at a particular level. This identifier has a value between 0 and N. It should be noted that N may have a different value at each particular level.

From the flowchart in Figure 4.10, it can be seen that the level of the clustering algorithm is set to 0 (this clustering level should not be confused with the level of a particular solution). Also, the new_sols parameter, that indicated if new solutions where created is set to true to enter the main loop. When this loop is entered the new_sols parameter is set back to false.

Then the algorithm iterates over all solutions in the solution set. Because the first beam operates only on the solutions at the current level, the "Do Beam 1" routine is called only if the level of the selected solution is equal to level of the clustering algorithm. So the first beam operates on a single solution which is at this point the parent solution. From this parent solution, 0, 1 or more child solutions can be constructed. This is done by merging two clusters in the non-flagged set of this solution (the flagged set is not touched). If there is no cluster or just a single cluster in the flagged set, no child solutions can be generated. However if there are two or more clusters, the remaining all pairs of clusters are considered based on their affinity. The higher this affinity the more likely that merging of the two corresponding clusters will lead to a better solution. The best pairs are actually merged, and for each merged pair a new solution is generated. The level of these newly generated solutions will be one higher than the clustering level of it's parent.

If a parent solution generates one or more new solutions, the new_sols parameter is set to true. If not, nothing is done. So if one or more parent solutions generate new solutions, the main loop will be executed again after beam 2 is done and the level is increased.

In contrast to the first beam, the second beam operates on a solution set (while the first beam operates on a single solution). Secondly, the second beam operates on all the solutions in the solutions set independent of their level. And finally, the second beam only rejects previously generated solutions (while the first beam only generates new ones). In this beam all the newly generated solutions (from all parents at the previous level) are compared to each other and to the solutions created earlier on in the clustering process. This comparison is done according to the solution cost. So the second beam selects the globally best solutions (of the entire solution space).

The stop condition has already been partially explained. But it will be explained here a little further. The obvious stop condition implied by the flowchart is that the loop is ended when the non-flagged sets of all solutions only contain one cluster (or no cluster, if the initial solution had no non-flagged clusters). This is of course a hard stop condition because at this point it is impossible to cluster any further. Note that for each particular solution, depending on the number of clusters in the non-flagged set when the clustering is started, the point where only one cluster remains may come at a different level.

There is also a second more subtle stop condition. It relates to the cases when there are new solutions created by the first beam, but they all are rejected by the second beam, because their quality is substantially lower (costs are higher) than the previously generated solutions. This creates a very natural stop condition. Also, by tweaking the cost function such that desired features of final solutions get a low cost and undesirable features get a high cost, the properties of the final solutions can be controlled.

The solutions that are still in the solution set when the stop condition is met, are considered to be final solutions of the clustering process and they constitute the input to the last phase of the method which is the actual creation of the partial machines and their interconnections for each of the solutions.

4.3.4 Beam 1 of the Clustering process

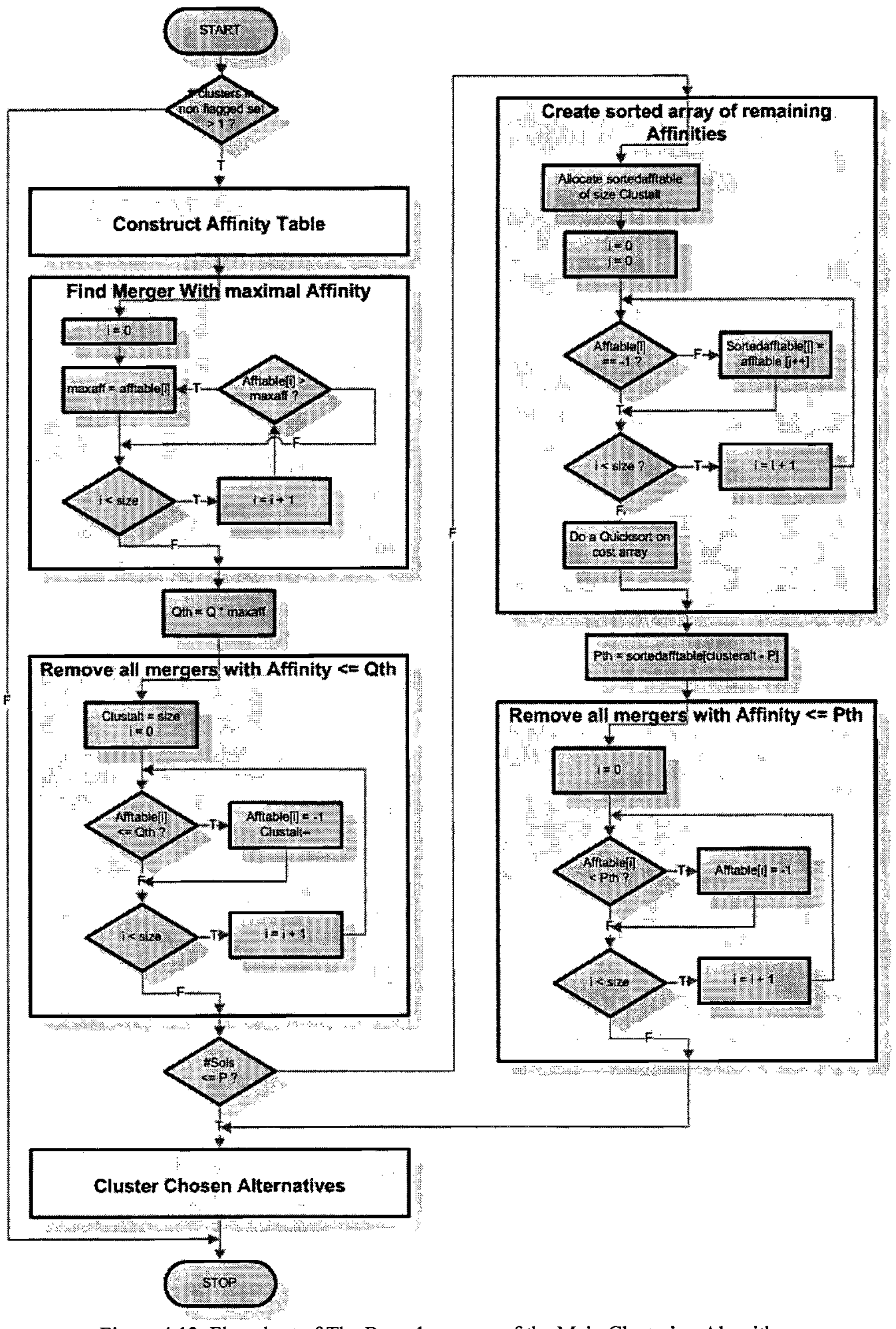
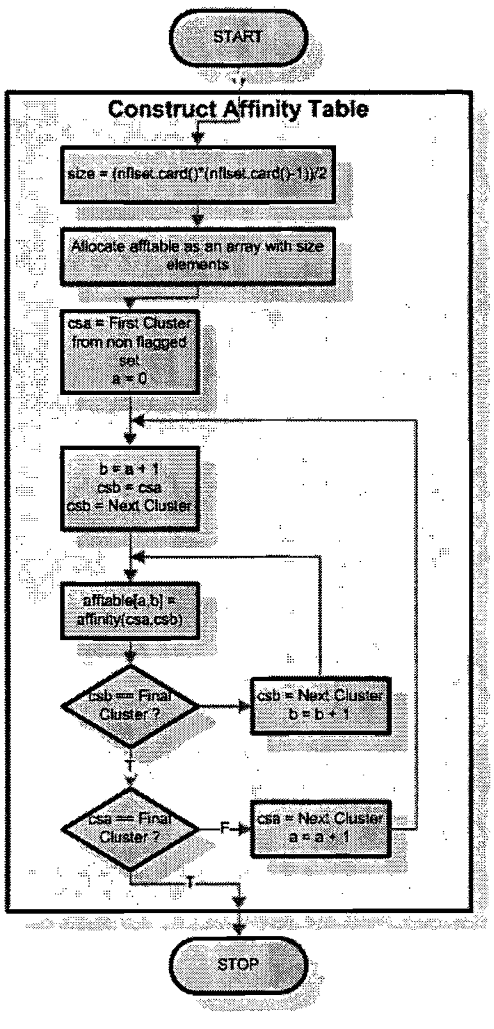

START
#clusters in non flagged set > 1 ?
T
F
Construct Affinity Table
Find Merger With maximal Affinity
i = 0
maxaff = afftable[i]
Afftable[i] > maxaff ?
T
F
i < size
i = i + 1
Qth = Q * maxaff
Remove all mergers with Affinity <= Qth
Clustalt = size
i = 0
Afftable[i] <= Qth ?
Afftable[i] = -1
Clustalt--
i < size
i = i + 1
#Sols <= P ?
Create sorted array of remaining Affinities
Allocate sortedafftable of size Clustalt
i = 0
j = 0
Afftable[i] == -1 ?
Sortedafftable[i] = afftable [j++]
i < size ?
i = i + 1
Do a Quicksort on cost array
Pth = sortedafftable[clusteralt - P]
Remove all mergers with Affinity <= Pth
i = 0
Afftable[i] < Pth ?
Afftable[i] = -1
i < size
i = i + 1
Cluster Chosen Alternatives
STOP

Figure 4.12: Flowchart of The Beam1 process of the Main Clustering Algorithm.


START
Construct Affinity Table
size = (nflset.card()*(nflset.card()-1))/2
Allocate afftable as an array with size elements
csa = First Cluster from non flagged set a = 0
b = a + 1 csb = csa csb = Next Cluster
afftable[a,b] = affinity(csa,csb)
csb == Final Cluster ?
csb = Next Cluster b = b + 1
T
csa == Final Cluster ?
F
csa = Next Cluster a = a + 1
T
STOP

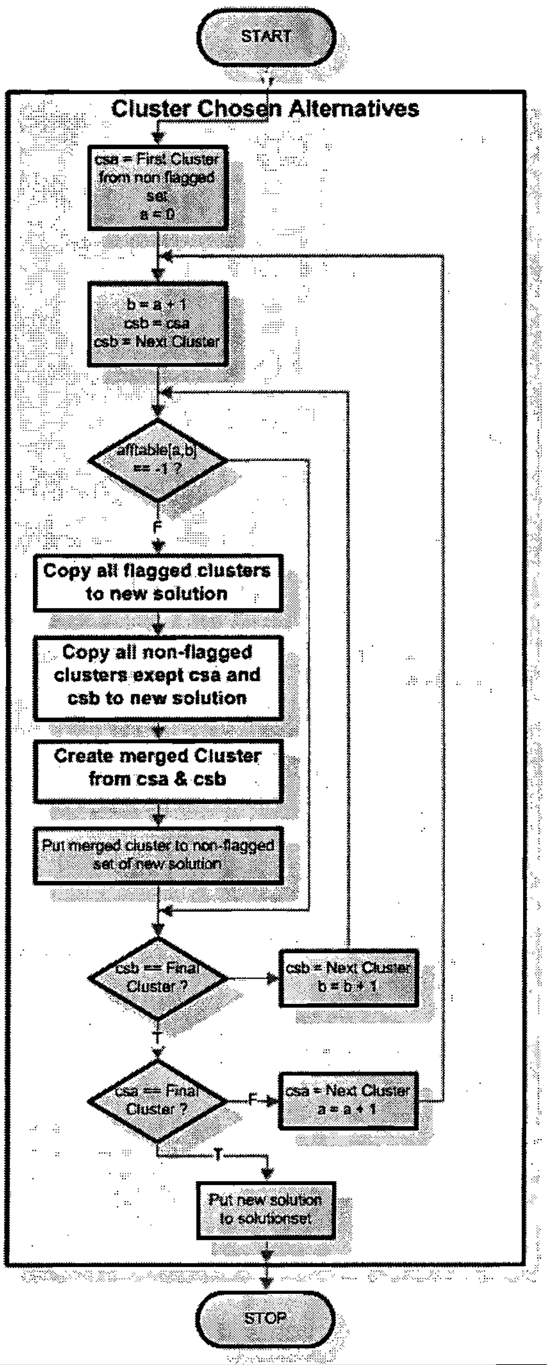

START
Cluster Chosen Alternatives
csa = First Cluster from non flagged set a = 0
b = a + 1 csb = csa csb = Next Cluster
afftable[a,b] == -1 ?
F
Copy all flagged clusters to new solution
Copy all non-flagged clusters exept csa and csb to new solution
Create merged Cluster from csa & csb
Put merged cluster to non-flagged set of new solution
csb == Final Cluster ?
csb = Next Cluster b = b + 1
T
csa == Final Cluster ?
F
csa = Next Cluster a = a + 1
T
Put new solution to solutionset
STOP

Figure 4.13: Flowchart of Construct Affinity Table.

Figure 4.14: Flowchart of Cluster Chosen Alternatives.

The first step of the first beam is to check if the non-flagged set contains more than one cluster. If this is not the case no mergers are possible for this solution and the beam is stopped.

If there is more than one solution in the non-flagged set, an affinity table is created, in which the affinity is stored for each pair of clusters in the non-flagged set. Note that affinity(a, b) = affinity(b, a). How the affinity for each pair of clusters is calculated is described below.

For each two molecules, it is possible to define and calculate an affinity measure related to each characteristic parameter of the two molecules, using the following equations:

Input Support Affinity Measure

$$\alpha_u(u_1, u_2) = \frac{|u_1| + |u_2| - |u_1 * u_2|}{\min(|u_1|, |u_2|)}$$

Output Support Affinity Measure

$$\alpha_v(v_1, v_2) = \frac{|v_1| + |v_2| - |v_1 * v_2|}{\min(|v_1|, |v_2|)}$$

Input Information Affinity Measure

$$\alpha_\alpha(\alpha_1, \alpha_2) = \frac{|IS(\alpha_1) \cap IS(\alpha_2)|}{\min(|IS(\alpha_1)|, |IS(\alpha_2)|)}$$

Output Information Affinity Measure

$$\alpha_\beta(\beta_1, \beta_2) = \frac{|IS(\beta_1) \cap IS(\beta_2)|}{\min(|IS(\beta_1)|, |IS(\beta_2)|)}$$

Pi Affinity Measure

$$\alpha_\pi(\pi_1, \pi_2) = \frac{|IS(\pi_1) \cap IS(\pi_2)|}{\min(|IS(\pi_1)|, |IS(\pi_2)|)}$$

Rho Affinity Measure

$$\alpha_\rho(\rho_1, \rho_2) = \frac{|IS(\rho_1) \cap IS(\rho_2)|}{\min(|IS(\rho_1)|, |IS(\rho_2)|)}$$

Cross-tau Affinity Measure

$$\alpha_\tau(\tau_1, \rho_2) = \frac{\sum_{\{s_i, s_j\} \in Inf(\tau_1) \cap Inf(\rho_2)} w_{\{s_i, s_j\}}}{\sum_{\{s_i, s_j\} \in Inf(\tau_1)} w_{\{s_i, s_j\}}}$$

Where:

$$w_{\{s_i, s_j\}} = \frac{n - k}{n - 1}$$

n is the total number of clusters,
k is the number of clusters that:

$$\{s_i, s_j\} \in Inf(\rho_k)$$

The affinity measures as defined above have to be combined into one general affinity measure for each two arbitrary clusters. Because a normalized affinity measure is necessary, all partial affinities and global affinity must be in the range <0, 1>. If the global affinity is equal to 0, it means that clusters are not using or producing any common information (neither input support, nor produced and imported state information are the same). The opposite case, if affinity is equal 1, means that clusters are identical. Results closer to 1 are favorable, and indicate that the corresponding molecules should be clustered. The global affinity measure is a combination of the weighted affinity measures corresponding to particular molecule attributes, i.e.:

- input support affinity
- output support affinity
- input information affinity
- output information affinity
- produced state information affinity
- exported state information affinity
- imported state information affinity

The global affinity measure is expressed by the following equation:

Affinity (cluster a, cluster b) = (Wis*Affinity_is(a,b) + Wos*Affinity_os(a,b) +
Walfa*Affinity_alpha(a,b) + Wbeta*Affinity_beta(a,b) +
Wpi*Affinity_pi(a,b) + Wrho*Affinity_rho(a,b) +
Wtau*Affinity_tau(a,b)) / NofAffinities

Where:
Wx – Weight of each particular affinity x
NofAffinities – number of partial affinities
Affinity_x - a normalized partial affinity related to attribute x

After the affinity table is constructed all affinities are calculated, so there is a quality measure for all clustering alternatives (as shown in the left picture of figure 15 for a non-flagged set containing 5 clusters).

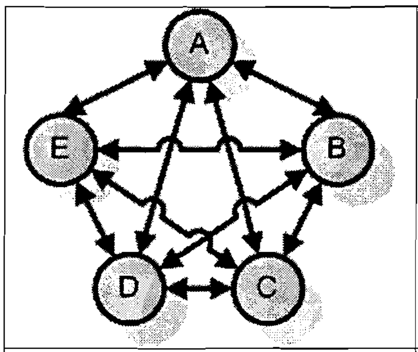
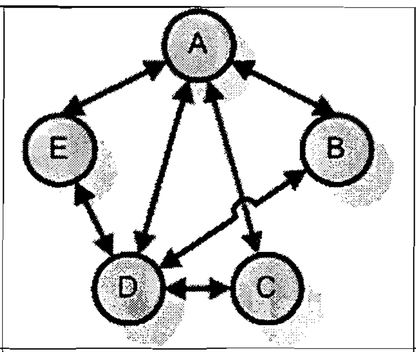
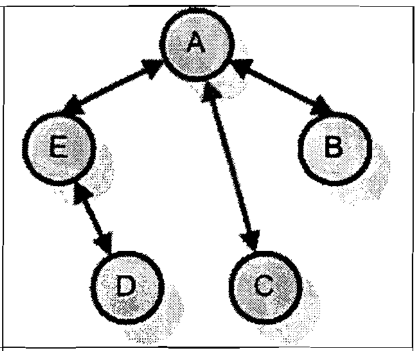
		
All alternatives	Remaining alternatives Quality	Remaining alternatives Quantity

Figure 4.15: Selection of clustering alternatives.

Because it may be too time consuming to consider all possible alternatives for all solutions, only the most promising (with the highest quality / affinity) alternatives are explored. First, the best clustering alternative is found. Then depending on Q, the quality factor, Qth, the quality threshold is determined. Q should be between 1 (accept only the best merger) and 0 (accept all mergers), and can be set as a constant or dynamically (e.g. as a function of the clustering level or the number of possible mergers from a particular parent) to maximize the probability that the search is steered towards the direction in the search space where these good solutions are. After Qth is determined, all merger alternatives with an affinity below Qth are rejected. This could lead to the situation in the middle picture of Figure 15.

If the number of remaining clustering alternatives (in case of our example 6) is lower than parameter P, the beam is finished. However if, to reduce computation time, we set P lower than the number of the remaining clustering alternatives, further selection is needed. This parameter P is an integer and can be (similar to Q) set as a constant of dynamically for a smarter search. If P is set to 1 only the best alternative is considered. If P = n, at most the n best alternatives are considered (if the beam is not limited by quality).

To realize this, the remaining cluster alternatives are sorted (with a quick search algorithm) from low to high. If we set the new threshold of the affinity based on the maximal number of allowable mergers (Pth) at the value of the sorted table with index clustering alternatives – P, we know below which value we should reject the clustering alternative to end up with at most P alternatives. An example of this can be seen in the right picture of Figure 8, for the case when we would accept at most 4 alternatives.

Once the candidates for clustering are selected, the actual child solutions are created. This is illustrated by the flowchart in Figure 7. For each particular accepted merger a solution is created. As explained in Chapter 4.3 a solution consists of a flagged and a non-flagged cluster set, a clustering level and a cost. The clustering level is one higher than the clustering level of the parent. The flagged set of the child solution can directly be copied to the flagged set of the child. The non-flagged set can also be copied except for the two clusters that are merged. Than a new cluster is made that produces (at least) the same output and state information as the two separate clusters before they where merged. Once the solution is complete the new cost can be calculated. How this is done exactly is described in Chapter 4.6, because this cost is used as the quality measure for the second beam.

4.3.5 Beam 2 of the Clustering process

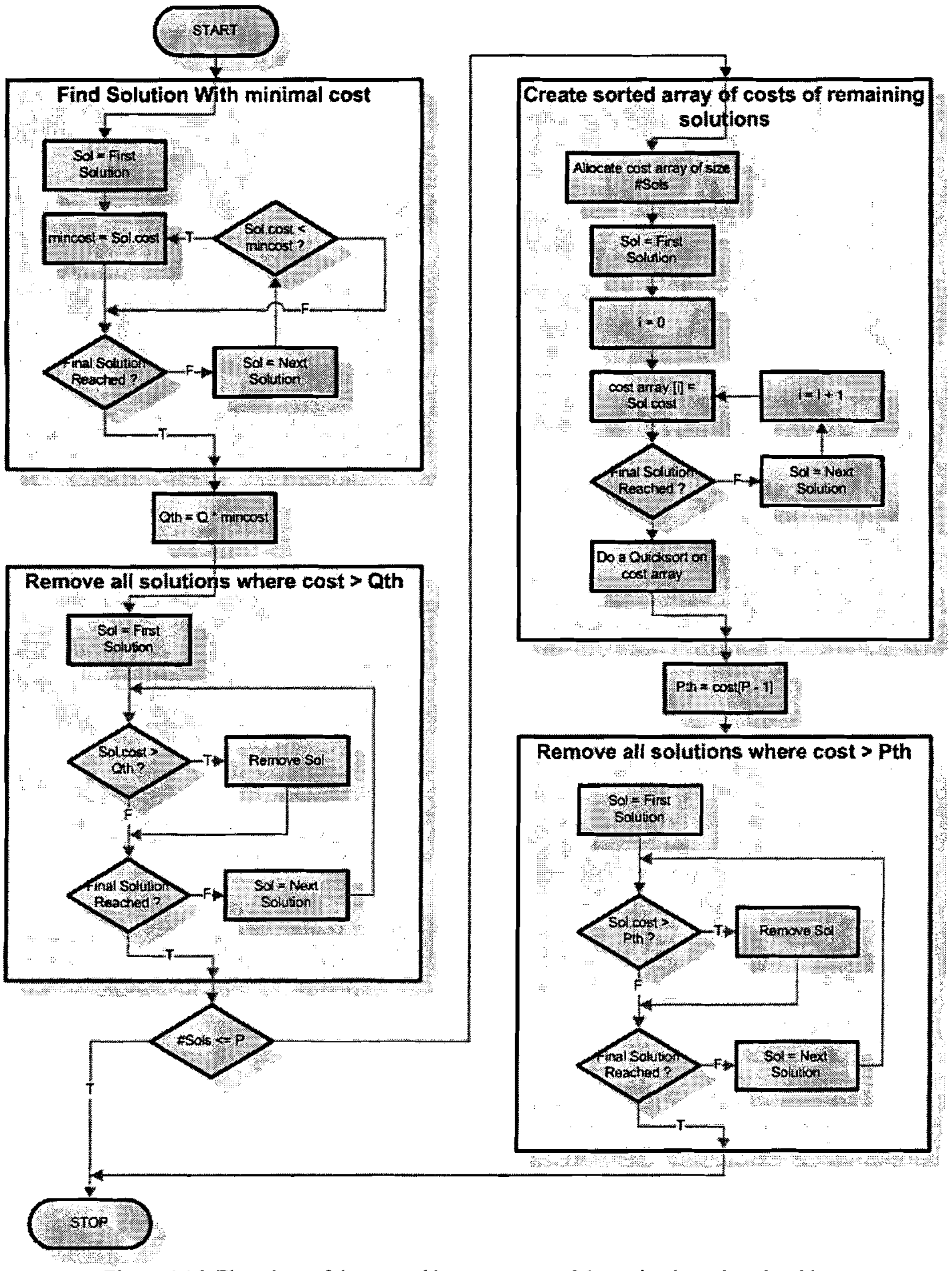

START
Find Solution With minimal cost
Sol = First Solution
mincost = Sol.cost
Sol.cost < mincost ?
T
F
Final Solution Reached ?
Sol = Next Solution
Qth = Q * mincost
Remove all solutions where cost > Qth
Sol.cost > Qth ?
Remove Sol
#Sols <= P
Create sorted array of costs of remaining solutions
Allocate cost array of size #Sols
i = 0
cost array [i] = Sol.cost
i = i + 1
Do a Quicksort on cost array
Pth = cost[P - 1]
Remove all solutions where cost > Pth
Sol.cost > Pth ?
STOP

Figure 4.16: Flowchart of the second beam process of the main clustering algorithm.

As described in Chapter 4.4, the second beam operates on all solutions (independent of their clustering level) and it doesn't create new solutions (it only removes solutions that are not good enough compared to solutions created from the solutions at previous levels and solutions created from different parents). The exact implementation is shown in the flowchart in Figure 9.

The second beam operates on a solution set (as opposed to beam 1 which operates on a solution). So for all solutions created so far the cost is asked (the cost is only calculated once during the creation of the solution). This cost function is calculated as follows:

$$\text{cost} = \left\{\left\lceil \log_2\left(|\beta|\right)\right\rceil + \left\lceil \log_2\left(|\rho|\right)\right\rceil\right\} \cdot cfunc\left\{\left\lceil \log_2\left(|\alpha|\right)\right\rceil + \left\lceil \log_2\left(|\pi|\right)\right\rceil, 4\right\}$$

where: $|\gamma|$ is the number of blocks on the γ set system, so $\left\lceil \log_2\left(|\gamma|\right)\right\rceil$ is the number of bits needed to encode the information from γ on. Here γ can be α (input information), β (output information), π (consumed state information) or ρ (produced state information).

Thus, $\left\lceil \log_2\left(|\alpha|\right)\right\rceil + \left\lceil \log_2\left(|\pi|\right)\right\rceil$ and $\left\lceil \log_2\left(|\beta|\right)\right\rceil + \left\lceil \log_2\left(|\rho|\right)\right\rceil$ represents the number of input and output bits to the combinational logic of a particular partial machine, respectively.

The cost of a combinational function with n input bits and 1 output bit is estimated by:

$$cfunc(n,k) = \begin{cases} 1 \text{ if } n \le k \\ \frac{1}{8}n^2 \text{ if } n > k \end{cases}$$

Here k is the number of input bits to one lookup table (usually 4). So, if the number of input bits is lower or equal to k, only one lookup table is used. If a function does not fit in a single lookup table the number of lookup tables is $c \cdot \frac{1}{2}n^2$ because of the triangular shape of the mapping of larger functions on smaller lookup tables where $c \approx \frac{1}{4}$. For a combinational function with more outputs the cost is simply multiplied by the number of output bits. It is clear that an assumption is made here that the outputs do not share substantial common logic. This assumption is however quite will satisfied, especially in the state logic of the minimally encoded FSMs.

This cost function is overestimating the cost for larger molecules, however this is no problem, because good decompositions consists of smaller molecules. E.g. if it is desired for the clustering process to produce even smaller (but probably more) partial machines, we could change cfunc to be $\frac{1}{8}n^3$ if $n > k$. This way, large partial machines become even more expensive.

First, the best single solution is determined. The best solution is the solution with the lowest cost. Because the solution with the lowest cost at a particular level does not have to lead to the best (or a very good) final solution, more solutions have to be considered. Since a low cost at a particular clustering level is an indication that it could lead to a good final solution, a limited number of the best solutions are accounted for. How many solutions exactly are considered can be controlled by the parameters P and Q (of the second beam). These can be set to a constant value but they could also be calculated dynamically (e.g. as a function of the clustering) to make better use of the computation resources because in this way the direction of the search can be steered.

The main selection criterion for accepting a particular solution is its quality. As stated earlier, in the second beam, the quality is based on the total solution cost. All solutions with a cost below the Quality threshold (Qth) are accepted. This Quality threshold is determined by the Q parameter and the Quality of the best solution. Q should be larger than 1, so this factor determines how much worse a solution may be and still be accepted by the algorithm.

If the most promising solutions exist of a comparable quality, it could to acceptance of too many solutions are accepted if only the quality parameter is used. Therefore another parameter P is used, which gives the maximal number of the accepted solutions by this beam. If the number of the accepted solutions is lower than P, the beam is finished. If however the number of the accepted solutions is larger than P, some of the worst of the accepted solutions should still be rejected such that P (or less) solutions remain.

To do this, an array is made of the costs of the remaining solutions. These are sorted (by a QuickSort algorithm) from low to high. So the element at index P-1 gives the new threshold Pth (because the first element is at index 0). Again all solutions above the new threshold (Pth) are rejected. This guaranties that no more than P solutions are accepted.

5. Experimental research

Although the whole method is based on the underlying analytic theories of general decomposition, information relationships and measures and set systems, many of the algorithms developed involve some heuristics. This is due to the fact that the problems to be solved here are to large and to complex by nature to be solved purely analytically. Also, no search other than exhaustive guarantees strictly optimal results. This is not only due to the complex combinatorial nature of this problem, but also because it is very hard to describe mathematically what "optimal" actually means. Therefore, the only way to proof the actual effectiveness of the SeMaDe tool with its underlying methods and algorithms, and if application of the theory of general decomposition and information relationships is successful is by experimental research.

For the testing of the first two parts of the SeMaDe tool (FSM information flow analysis and clustering) many benchmarks were used. For this purpose we had 3 sources of benchmarks. The first was the MCNC set of industrial benchmarks. This set consists of a set of about 40, most of them, relatively small benchmarks. Secondly, we used a set of approximately 150 benchmarks generated with the benchmark generator. This set covers a major part of all different classes of sequential machines (Moore and Mealy machines; state-, input-, output-dominated machines; and of different sizes: small, medium and large machines). Finally we used a set of about 50 Moore machines also generated by the BenGen tool. This set was created because not all parts of the implementation (molecule sub-set selection) have been extended to Mealy machines yet. However in Chapter 4, a reasonable detailed description is given on how this extension for Mealy machines could be made.

It is not yet possible to perform a more extensive experimental research on the entire method, because the final part of the method still has to be implemented. Once this is done, automated testing and experimentation can be performed and its results can be compared to those of other commercial and academic tools. For now, we can already say that results are very promising by analyzing the log-files from the tool. However analyzing these log files involves a lot of tedious work which has to be done by hand. Secondly, it does not allow for an easy comparison to other tools.

In this chapter we will explain for one very simple machine the steps performed by the algorithm and we will discuss the resulting decompositions generated by the tool. The entire log-file is not presented here, since it is too long even for these small machines.

5.1 Example

As mentioned in Chapter 4, the FSM specification is read from a KISS (.exl) file. Below, an example of such a KISS file is shown:

```
.i 4
.o 5
.p 19
.s 8

00--   st1   st2   00000
10--   st1   st1   00000
11--   st1   st8   00000
01--   st1   st1   00000
---0   st2   st3   00100
---1   st2   st2   00100
-0--   st3   st2   01001
-1--   st3   st4   01001
-0-1   st4   st3   01101
-1--   st4   st5   01101
-0-0   st4   st4   01101
1--0   st5   st4   11011
---1   st5   st6   11011
0--0   st5   st5   11011
----   st6   st7   11111
----   st7   st8   10001
-1-0   st8   st7   10101
-0--   st8   st1   10101
-1-1   st8   st8   10101
.e
```

In the beginning of this file, the number of inputs, outputs, product terms (or transitions) and states are given (preceded by the .i, .o, .p and .s tokens, respectively). Subsequently, at each new line a transition is specified using the following four columns: input cube and present state followed by its corresponding next state and output cube. The .e token marks the end of the file.

Once the KISS file is read by SeMaDe, a dictionary is created that assigns a symbol (as an integer) to each input cube, state and output cube :

```
Input Dictionary {        State Dictionary {        Output Dictionary {
  (0)  00--                 (0)  st1                  (0)  00000
  (1)  10--                 (1)  st2                  (1)  00100
  (2)  11--                 (2)  st8                  (2)  01001
  (3)  01--                 (3)  st3                  (3)  01101
  (4)  ---0                 (4)  st4                  (4)  11011
  (5)  ---1                 (5)  st5                  (5)  11111
  (6)  -0--                 (6)  st6                  (6)  10001
  (7)  -1--                 (7)  st7                  (7)  10101
  (8)  -0-1               }                         }
  (9)  -0-0
  (10) 1--0
  (11) 0--0
  (12) ----
  (13) -1-0
  (14) -1-1
}
```

From the KISS specification it can be seen that only one output combination occurs for each particular current state. This indicates that the machine is a Moore machine. Further, after a closer analysis of the output dictionary, it can be seen that outputs 3 and 4 (in the 4th and 5th column) can be constructed as simple functions (AND, OR) from the first three outputs using output decoders.

First, the SeMaDe tool starts by creating a Mealy A molecule for each particular output. Since the machine considered is a Moore Machine, none of these Mealy A molecules requires input information.

Therefore, the SeMaDe tool recognizes that this is a Moore machine and also constructs five Moore A molecules corresponding to each of the five outputs. The cost for these molecules are 4.5 for output 0, 1.0 for output 1, 4.5 for output 2, 1.0 for output 3 and finally 3.125 for output 4.

Subsequently, the maximal SP set system is computed. Because all the constructed A molecules together consume all the state information, also all the state information (0-set system) has to be produced by the B molecules. This information is covered by seven B molecules.

Then the molecule sub-set selection algorithms starts. Using Q factors of 1.25 and 1.23 for the first and second beam, respectively, the sub-set selection algorithm finds the following three solutions:

```
parentset(after beam): MooreSelSet {
  MooreSel Mooresupport: { 0, 1, 2 } Bsupport: { } cost: 10,
  MooreSel Mooresupport: { 1, 2, 3, 4 } Bsupport: { } cost: 9.625,
  MooreSel Mooresupport: { 0, 1, 2, 3 } Bsupport: { } cost: 11
}
```

Since none of the constructed solutions require any B molecules to make the decomposition valid, clustering is trivial (no actual clustering will be performed). Clustering only rejects the third solution, resulting in:

```
after beam 2: SolutionSet {
  Solution (level: 0 and cost: 9.625{
    Flagged ClusterSet:   ClusterSet {
      Cluster (5) cost: 3.125 "M4" { }
      Cluster (6) cost: 1 "M3" { }
      Cluster (7) cost: 4.5 "M2" { }
      Cluster (8) cost: 1 "M1" { }
    }

    Non Flagged ClusterSet: ClusterSet {
    }
  },

  Solution (level: 0 and cost: 10{
    Flagged ClusterSet:   ClusterSet {
      Cluster (7) cost: 4.5 "M2" { }
      Cluster (8) cost: 1 "M1" { }
      Cluster (9) cost: 4.5 "M0" { }
    }

    Non Flagged ClusterSet: ClusterSet {
    }
  }
```

The second solution actually is the expected solution, where three partial machines are created for outputs 0,1 and 2 and where outputs 3 and 4 are constructed by (at this point still virtual output decoders). Also these three molecules appear to have SP property for the state information, which make this decomposition valid (this cannot be easily analyzed by a human designer).

But also a second cheaper solution that is not so obvious for a human designer was constructed by SeMaDe. In this solution also Moore molecules are constructed for outputs 1 and 2, but instead of a Moore molecule for output 0, two Moore molecules for output 4 and 5 are constructed. As it appears, these two molecules together have a lower cost than the single molecule for output 0. This is mainly due to a large amount of state information consumed by the molecule for output 0. Also, output 0 can be computed (using a virtual output decoder) from the outputs 1 to 4. So this second solution actually is (but only just) superior to the expected one. However, it is still useful to consider both solutions during the rest of the method (or even until after combinational logic synthesis), because their estimated quality is almost identical, but one of them may e.g. have a difficult to implement sub-function.

6. Conclusion

The subject of the master project reported here was the FSM information flow analysis for general decomposition that constitutes the first part of the larger FSM architecture synthesis method briefly described in this report.

The main aim of this master project was to develop several new or modified better algorithms for the FSM information flow structure analysis part of the SeMaDe tool and their actual software implementation in C++. This also accounted for the final adaptation of the software developed for the second (clustering) part of the method to seamlessly collaborate with the first part. This aim is achieved. The algorithms have been developed or modified and implemented. The results are documented mainly in Chapter 4 of this report.

In particular, the modified efficient algorithms for the molecule computation have been developed and implemented. These algorithms effectively exploit the algebraic lattice structure of the computation problems to efficiently perform the computations. The modified and new analysis algorithms perform computation of the most difficult to compute information much more efficient then the original algorithms. They also detect some relevant special cases (i.e. Moore type outputs).

Moreover, in a few places of the FSM information flow analysis algorithm, a double beam search is implemented. In this way many good quality solutions can efficiently be examined and further developed in parallel by the algorithm instead of just a single one. This greatly increases the quality of the final results and the robustness of the method.

Also a new efficient algorithm was developed (based on clique splitting) for very fast conversion between an information set and a set system. This gives the method the ability to not only take the actual information into account, but also its structure, which is of major importance, because this is needed, for instance, to determine the actual number of bits to transport the information from the partial machines to the terminals (inputs and outputs), with or without input encoders and output decoders and also between the partial machines.

At the moment of finishing of this master project, the first two modified phases of the method, FSM information flow analysis and clustering, were completely described, implemented in a prototype SeMaDe tool and documented. This tool produces a log file giving information on how the tool handles the particular decomposition steps for a certain specification of a sequential machine. We ran several hundreds of benchmarks to test the tool, and analyzed the log files of several benchmarks to check the quality of the produced decompositions. The extensive testing confirms that after the modifications performed, the first two parts of the SeMaDe tool, work correctly and produce valid decomposition structures (at least for all the hundreds of test cases). For all the more precisely analyzed cases, the tool produced very good results of a quality that was expected (for some of these machines a very good decomposition was known), and sometimes even some surprising / unexpected decompositions where found which were of comparable or even better quality that the known very good decomposition.

However, at this stage it is only possible to say that the method and its current implementation are very promising and the first two software parts seem to work as required. The final benchmarking cannot yet be performed, because the final part of the whole method is not yet implemented as required. Once the final phase, construction of a network of partial machines, is fully implemented, besides a log file, the tool will produce the actual FSM network specification in the formats understandable to the sequential and combinational logic synthesis tools. Once this is achieved, the method can be compared to commercial and academic tools on many benchmarks. The benchmarking and analysis can be fully automated, which will make a much more extensive benchmarking and analysis possible. This will finally confirm if the method is successful.

Also there are still some further improvements possible to the algorithms and the tool. The analysis part could be complemented with some additional improvement of the molecules. Also this part is not standard in any way, so completely new insights on how to do this are still possible.

The clustering phase is a more standard process. It is implemented in a very general way with a very natural stop condition. Here however, a further improvement can also be made, e.g. by defining possibly better types of affinities, cost function, weights and beam parameters. Also, these different parameters can be changed at run-time (made dynamic), or changed to get particular results (e.g. optimize for speed, area, power consumption), or to control the size of the search, the number and size of the partial machines, etc. The adjustment of these functions and parameters can also be done by learning algorithms (like in neural networks or fuzzy systems), which can use the extensive set of benchmarks as learning data. All these possible further works were however not in the scope of this master project.

References

[1] A. Chojnacki: Effective and Efficient Circuit Synthesis for LUT FPGAs Based on Functional Decomposition and Information Relationships Measures, Ph.D. Dissertation, ISBN 90-386-1673-2, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands, 2004, pp. 1-286.

[2] A. Chojnacki, L. Józwiak: An Effective and Efficient Method for Functional Decomposition of Boolean Functions Based on Information Relationships Measures, Design and Diagnostics of Electronic Circuits and Systems DDECS'2000, Smolenice, Slovakia, April 5-7, 2000.

[3] J. Hartmanis: Symbolic Analysis of a Decomposition of Information Processing, Information and Control, vol. 3, pp. 154-178, June 1960.

[4] J. Hartmanis: Loop-free Structure of Sequential Machines, Information and Control, vol.5 , pp. 25-43, 1962.

[5] J. Hartmanis, R.E. Stearns: Algebraic Structure Theory of Sequential Machines, Englewood Cliffs, N.J.: Prentice-Hall, 1966.

[6] L. Jóźwiak: General Decomposition and Its Use in Digital Circuit Synthesis, VLSI Design: An International Journal of Custom Chip Design Simulation and Testing, vol.3, No 3-4, 1995.

[7] L. Jóźwiak: Information Relationships and Measures - An Analysis Aparatus for Efficient Information System Synthesis, Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97), Budapest, Hungary, September 1-4, 1997, pp. 13-23., IEEE Computer Society Press.

[8] L. Jóźwiak: Advanced AI Search Techniques in Modern Digital Circuit Synthesis, Artificial Intelligence Review, ISSN 0269-2821, Kluwer Academic Publishers, Dordrecht, The Netherlands, Vol. 20, No 3-4, December 2003, pp. 269-318.

[9] L. Józwiak, A. Chojnacki: Effective and Efficient FPGA Synthesis through Functional Decomposition Based on Information Relationship Measures, DSD'2001 - Euromicro Symposium on Digital System Design, September 4-6, 2001, Warsaw, Poland, ISBN 0-7695-1239-9/01, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 30 - 37.

[10] L. Jóźwiak, A. Chojnacki: Effective and Efficient FPGA Synthesis through General Functional Decomposition, Journal of Systems Architecture, ISSN 1383-7621, Elsevier Science, Amsterdam, The Netherlands, Vol. 49, No 4-6, September 2003, pp. 247-265.

[11] L. Jóźwiak, D. Gawlowski and A. Ślusarczyk: An Effective Solution of Benchmarking Problem - FSM Benchmark Generator and Its Application to Analysis of State Assignment Methods, DSD'2004 - Euromicro Symposium on Digital System Design, August 31st - September 3rd , 2004, Rennes, France, ISBN 0-7695-2003-0, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 160 - 167.

[12] L. Jóźwiak, D. Gawłowski, A. Ślusarczyk: Multi-objective Optimal FSM State Assignment, DSD'2006 – 9th Euromicro Conference on Digital System Design, August 30 - September 1, 2006, Cavtat near Dubrovnik, Croatia, ISBN 0-7695-2443-8, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 385-393.

[13] L. Jóźwiak, D. Gawłowski, A. Ślusarczyk: Multi-objective Optimal Controller Synthesis for Heterogeneous Embedded Systems, IC-SAMOS'2006 – International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, July 17-20, 2006, ISBN 1-4244-0155-0, IEEE Press, Piscataway, NJ, USA, pp. 177-184.

[14] L. Jóźwiak, D. Gawłowski, A. Ślusarczyk: Benchmarking in Electronic Design Automation, MIXDES'2006 – International Conference on Mixed Design of Integrated Circuits and Systems, Gdynia, Poland, June 22-24, 2006, ISBN 83-922632-1-9, DMCS, Lodz, Poland. pp. 245-250.

[15] L. Józwiak, P.A. Konieczny: Heuristic Algorithms for Minimal Input Support Problems, International Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Slovakia, 11-13 Sept., 1995.

[16] L. Józwiak, A. Slusarczyk: Application of Information Relationships and Measures to Decomposition and Encoding of Incompletely Specified Sequential Machines, Third Oregon Symposium of Logic, Design and Learning, Oregon, Portland, USA, May 22, 2000.

[17] L. Jóźwiak, A. Ślusarczyk: General Decomposition of Incompletely Specified Sequential Machines with Multi-State Behavior Realisation, Journal of Systems Architecture, ISSN 1383-7621, Elsevier Science, Amsterdam, The Netherlands, Vol. 50, December 2003, pp. 445-492.

[18] L. Jóźwiak, A. Ślusarczyk, M. Perkowski: Term Trees in Application to an Effective and Efficient ATPG for AND-EXOR and AND-OR Circuits, VLSI Design: An International Journal of Custom Chip Design Simulation and Testing, vol. 14, No 1, April 2002.

[19] P.A. Konieczny: General Decomposition of Sequential Machines – Algorithms and Programs Eindhovense School voor Technologisch Ontwerpen, IVO. – III. ISBN 90-5282-469-X

[20] P.A. Konieczny and L. Jóźwiak: Minimal Input Support Problem and Algorithms to Solve It, Eindhoven University of Technology Research Reports, EUT Report 95-E-289, Eindhoven University of Technology, The Netherlands, April 1995.

[21] M. Rawski, L. Jóźwiak, T. Luba: Functional Decomposition with an Efficient Input Support Selection for Sub-functions Based on Information Relationship Measures, Journal of Systems Architecture, ISSN 1383-7621/01165-6074, Elsevier Science, Amsterdam, The Netherlands, 2001, Vol 47/2, pp 137-155.

[22] A. Ślusarczyk: Decomposition and Encoding of Finite State Machines for FPGA Implementation, Ph.D. Dissertation, ISBN 90-386-1663-5, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands, 2004, pp. 1-187.

[23] F.A.M. Volf: A Bottom-up Approach to Multiple-level Logic Synthesis for Look-up Table FPGAs, Ph.D. Dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997 (ISBN 90-386-0380-0).