Eindhoven University of Technology

MASTER

Development of a structured approach to the decomposition problem in digital system design

Slenders, P.J.W.

Award date:
1987

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

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

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

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Development of a structured approach to the decomposition problem in digital system design

By P.J.W. Slenders

Master thesis report by: P.J.W. Slenders
Coach: Prof. ir. M.P.J. Stevens

Eindhoven, The Netherlands
December 1987

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.
ABSTRACT

This report describes the development of a structured approach to the decomposition problem in digital system design. From the theoretical concepts of abstract automata, algorithms and parallel processes a general framework for the decomposition of digital systems is derived. As a result, some recommendations are made towards the design cycle of digital systems in general, and the decomposition phase in particular.

Headwords

Digital systems, hardware algorithms, decomposition, parallel processes, communication and synchronization
Acknowledgements

I would like to thank:

- Prof. ir. M.P.J. Stevens for motivating me when everything seemed to slip away, and for the pleasant support he gave me throughout the graduation period.

- my dear friend Frank Budzelaar for his infectious enthusiasm, his patience and, not in the least, his numerous suggestions.

- all other people who helped me during my study and who turned it into a period which is worthwhile to remember for a long time.

Eindhoven, November 1987
Peter Slenders
# Table of Contents

1. Introduction .......................................................................................................................... 5

2. Theoretical backgrounds ........................................................................................................ 9
   2.1 Abstract automata theory .................................................................................................... 9
   2.2 Classical decomposition theory .......................................................................................... 17
   2.3 Concept of algorithm ......................................................................................................... 20
   2.4 Correspondence with software design ................................................................................ 27

3. The development of the TUE decomposition model ................................................................. 29
   3.1 Glushkov's decomposition model ...................................................................................... 29
   3.2 Sample decomposition 1: a simple transmitter .................................................................. 33
   3.3 Consequences for Glushkov's model (1) ........................................................................... 45
   3.4 Sample decomposition 2: a CPU-controlled transmitter ..................................................... 48
   3.5 Consequences for Glushkov's model (2) ........................................................................... 55

4. Parallelism ................................................................................................................................ 59
   4.1 Introduction ......................................................................................................................... 59
      4.1.1 Sample decomposition 3: transmitter with parity ......................................................... 59
      4.1.2 Example 4: the multiple adder problem ...................................................................... 66
   4.2 Communication and synchronization .................................................................................. 67
   4.3 Consequences for Glushkov's model (3) ........................................................................... 72
      4.3.1 Example 5: the use of semaphores ............................................................................ 73

5. Decomposition of complex systems ....................................................................................... 87

6. Final contemplations ............................................................................................................... 92

7. Conclusion ............................................................................................................................... 96

Literature .................................................................................................................................... 97
1. Introduction

Since the early days of digital system design a trend has been seen in which the complexity of digital systems constantly increases. This progressive trend can be illustrated on basis of the growing complexity of integrated circuits. These circuits form nowadays the state of the art in digital system component technology, and the evolution of integrated circuits gives a good impression of the rising complexity of digital systems.

![Figure 1.1 Growth in complexity of integrated circuits](image)

Since its beginnings in 1962, integrated circuit technology has improved continuously, the most important consequences of this progress being \[\text{ANCEAU}\]:

- an annual increase in the maximum complexity of integrated circuits by a factor of between 1.6 and 2;

4
- a constant improvement in their operational speed of around 20% per year;
- a reduction in the cost per function by an annual factor of between 1.6 and 2.

This increase in complexity (shown in figure 1.1) can be mainly attributed to the improvements in lithographic techniques, but also the increase in the maximum chip size and the improvements in circuitry, giving a reduction in the number of transistors required to realize a given function, give rise to more complex integrated circuits.

Advanced studies show that this technological progress is by no means exhausted and that, because of very great economic pressure (very large-scale integrated circuits make manufacturing and maintenance groups tremendously productive), it will continue for at least another decade.

So, integrated circuit complexity, and thus also digital system complexity, continues to grow roughly exponentially, and the person-years to specify and design those chips are stretching out at a comparable or perhaps even faster rate. At the same time that the product development cycle is lengthening, the lifetime of products is getting shorter (see figure 1.2). The same increase in complexity that slows development accelerates the pace of improvements that can be gained from a new generation of designs. Those advantages of highly integrated systems also spur competition through opening new and lucrative markets, such as office automation.

Figure 1.2 Product lifetime versus product-development time

That combination of shorter product life and longer development time threatens the untenable situation of second-generation pro-
products being developed before a first generation has even tested its markets. In order to prevent this situation, some means have to be found by which the design cycle (product development cycle) of very complex digital systems can be shortened. In other words, we have to manage the ever growing complexity of digital systems in such a way that the time period needed to specify and design those systems can be reduced to, at least, an acceptable level.

For that purpose, at the Eindhoven University of Technology, a research group under supervision of Prof. Stevens is working on the development of a methodology for capturing the growing complexity in digital system design and designing fault-free digital systems in an, as much as possible, optimal manner.

Starting points for this research project are the following ideas:

- the most important way to get control over the complexity of a system, is to make optimal use of the intrinsic hierarchical character of the system;

- the design cycle has to be structured, in order to be able to use computer tools as a means to manage the increasing complexity.

In this project, the design cycle of digital systems is the central research subject. Globally this design cycle can be described as the following sequence:

\[ \text{requirements and specification phase} \]
\[ \downarrow \]
\[ \text{formal specification phase} \]
\[ \downarrow \text{decomposition} \]
\[ \text{implementation phase} \]
\[ \downarrow \text{realization phase} \]

Figure 1.3 The design cycle of digital systems

In the first step, the requirements and specification phase, the requirements of the system are determined, and on the basis of this information a first and informal specification is derived. This informal specification is formalized in the second step, the formal specification phase. The result of this phase is an exact and unambiguous specification of the desired system. Usually the behavior of the system is specified in an appropriate behavior-language.
Next, the formal specification is decomposed a number of times until a description of the system in terms of realizable building blocks, for instance gates, macro-cells or IC's, has been derived. This logical network is called the implementation of the system. Finally, the system, i.e. the network of building blocks, is physically realized, e.g. in the case of an integrated circuit the chip is produced, with the help of the lay-out masks. Another example of this realization phase is the following: in the case a system is built up out of discrete components, the realization phase involves the placement of the components on the printed circuit boards.

The decomposition of a system into a number of subsystems, each of which can be designed more easily, forms a very important step in the design cycle. The gradual introduction of detailed information helps us to manage the complexity of the system in a hierarchical way. However, the unstructured character of this procedure places us for a problem. At the moment the skills of the designer, his insight and experience, are the most important factors which have influence on the quality of the decomposition. These elusive elements, and the corresponding irreproducibility of the decomposition, make the use of computer tools difficult. At this moment it seems to be worthwhile to investigate the decomposition problem in general, and the possible structuring of it, more intensely. In this report an account is given of such an investigation of the decomposition problem.

In chapter 2 some important theoretical backgrounds of digital systems are presented. On the basis of this information a very important decomposition model is derived, and some of its properties are studied (chapter 3). Next, in chapter 4, the decomposition model is extended with the concept of parallel processes, and in chapter 5 the decomposition of complex systems is investigated. Finally, in chapter 6, some concluding remarks and recommendations are made.
2. Theoretical backgrounds

In this chapter some theoretical backgrounds, which lie at the basis of digital systems in general, and thus of the decomposition problem in digital system design in particular, will be introduced. It will be shown that the concept of algorithm is the most fundamental theoretical basis of digital system design.

In order to decompose complex digital systems one can follow one of two possible methods: first it is possible to pick ad-hoc an arbitrary set of rules which present a solution to the decomposition problem or, in the second place, a set of rules can be derived from the knowledge about the mathematical backgrounds of digital systems.

In this report an attempt is made at finding such a decomposition method based upon knowledge about the mathematical backgrounds of digital systems. In the next paragraphs the exact mathematical basis of digital systems will be established.

2.1 Abstract automata theory

In this chapter a notion of the theory of abstract automata and its coherence with digital systems is presented. In no way is the presentation of these subjects exhaustive: it is just tried to give the reader a feeling of understanding, not a feeling of exact knowledge.

A general model of an information processing system, that produces some output signal (a letter of the output alphabet) in response to each input signal (a letter of the input alphabet), is given by the concept of abstract automata.

Definition:

An automaton is a quintuple $M=\langle X, Y, Q, s, o \rangle$, where

$X$ is the set of inputs;
$Y$ is the set of outputs;
$Q$ is the set of states;
$s$ is the next-state function;
o is the next-output function.

An abstract automaton is a general model of an information processing system, considered without regard to its internal structure.
To specify an abstract automaton we must specify three sets, an input alphabet \( X \), an output alphabet \( Y \) and a set of internal states of the automaton, which we shall denote by the letter \( Q \). The automaton operates in discrete time, successive instants of which are conveniently identified with successive natural numbers \( t=0,1,2,... \) (this can always be done with a suitable unit of time). At each instant of discrete automaton time \( t=0,1,2,... \), the automaton \( M \) is in one definite state \( q=q(t) \) from the set \( Q \) of its internal states. The state \( q_0=q(0) \) at the initial instant of time \( t=0 \) is called the initial state of the automaton \( M \). At each instant of time \( t \) beginning at \( t=1 \) the automaton receives as an input signal one of the letters \( x=x(t) \) from the input alphabet \( X \). Finite ordered sequences of input signals \( x(1)x(2)...x(k) \) of the automaton are called the input words of this automaton. The input of the automaton may receive any input word from some fixed set of permissible input words. Any permissible word \( X_w=x(1)x(2)...x(k) \) applied to the input of an automaton \( M \) produces an output word \( Y_w=y(1)y(2)...y(k) \) which is some ordered finite sequence of output signals of the automaton \( M \) (letters of its output alphabet \( Y \)) that has the same length as the corresponding uniquely defined input word \( X_w \). The obtained correspondence \( f \) between the permissible input words \( X_w \) and the corresponding output words \( Y_w \) is called an (alphabetic) mapping induced by the automaton \( M \).

The mapping \( f \) is uniquely defined by the specification of two functions \( s \) and \( o \), called, respectively, the transition function and the output function of the automaton. The transition function defines the state \( q(t) \) of the automaton at any instant of discrete time \( t \) according to the input signal \( x(t) \) at the same instant, and the state \( q(t-1) \) at the previous instant of time:

\[
q(t)=s(q(t-1);x(t)) \quad (1)
\]

The output function defines the output signal \( y(t) \) as the function of the same variables:

\[
y(t)=o(q(t-1);x(t)) \quad (2)
\]

Having specified any input word \( p=x(1)x(2)...x(k) \) and an initial state \( q(0) \) of the automaton, by means of relationships (1) and (2) we can subsequently determine all letters of the corresponding output word:

\[
q=f(p)=y(1)y(2)...y(k)
\]

Thus, relationships (1) and (2) effectively define the mapping \( f \) induced by the automaton \( M \).

This brief presentation of the theory of abstract automata would be of no use if it would not be possible to apply this theory in some way in the digital systems field. Fortunately, there is a class of
abstract automata, the so-called finite automata, which has a strong parallel with digital systems.

An automaton is called finite if all three of its defining sets $X$, $Y$ and $Q$ are finite. In the literature about digital systems this type of automaton is often called finite-state automaton or finite-state machine, which is in fact an incomplete indication of a finite automaton.

Globally speaking, digital systems can be divided into two categories: sequential circuits and combinational circuits. A digital system is called combinational if its outputs depend only on the present inputs, and a digital system is referred to as being sequential if its outputs depend on both the present inputs and the history of the inputs to the system. From this, it is immediately clear that a combinational circuit can be modelled by a finite automaton which has a single state only (the initial state). Furthermore in any practical sequential circuit the influence of past inputs to the present outputs is limited, e.g. the history of inputs is represented in a finite number of internal states. So, sequential circuits can also be modelled by a finite automaton which has a finite (>1) number of states.

Obviously, the concept of finite automata can be used to model digital systems. Even stronger, we can say that digital systems are in fact practical implementations of finite automata.

As was mentioned above, finite automata form a class of abstract automata. There exists in fact a hierarchy of abstract automata (figure 2.1). This hierarchy, which is known as Chomsky's hierarchy, indicates the problem-solving power of each class of abstract automata.

The relative power of each class of automata becomes clear when we adopt the following model: we can view an abstract automaton as a machine (a primitive computer) which has an input tape (or file) of cells, a corresponding read head, an output tape of cells, a corresponding write head, a work tape of cells, a corresponding read-write head and a finite control. The finite control has knowledge of the transition function $s$, the output function $o$ and the current state of the automaton. The input tape contains the input word, one symbol to each cell of the tape. This means that the tape has exactly the same number of cells as the input word has symbols. It can be thought of as a one-dimensional array of symbols. The output tape is used for storing the output word (one symbol to each cell). The work tape is used for storing and retrieving intermediate results, if necessary. We shall now exemplify this model by means of a finite automaton.
The finite automaton (figure 2.2) is initialized with an input word $x$ as follows:

1. $x$ is placed on the input tape one symbol to a cell;
2. The reading and writing heads are positioned over the leftmost cells in their respective tapes;
3. The current state is set to the start state $g$ (goal);
4. The finite automaton is started.

The finite automaton, once started, begins its computation on the input word. As with any computer it has a basic execution cycle:

1. The symbol under the reading head is read, that is, the current symbol. If there is no symbol under the reading head the finite automaton terminates. This occurs when the whole input word has been read;
2. The next state and the output are computed from the current state and symbol using the transition function.
Theoretical backgrounds

and the output function. If the current state is undefined the finite automaton aborts;
3) The reading head moves one cell to the right;
4) The next state becomes the current state and the output symbol is written to the output tape;
5) The writing head moves one cell to the right and the execution cycle has been completed.

The finite automata (Chomsky's type 3) can be used for solving a relative small class of problems.

Pushdown stores or stacks are one of the most frequently chosen data structures. They are used to evaluate arithmetic expressions in postfix form and to generate code for their evaluation; to keep track of the return addresses and environments invoked by subprogram calls, particularly recursive ones; to keep track of locally defined variables when executing block-structured languages. It should not be too surprising, therefore, that they form the basis of a fundamental and important automata-theoretic model.

A pushdown automaton is essentially a finite automaton with some additional machinery. First, we add an unbounded file or tape of symbols, the pushdown, and a read-write head for the pushdown; see figure 2.3. In order to simulate the stack data structure, only the topmost symbol can be read, that is the one before the end-of-file or eof, and a write is only allowed if the pushdown is at eof. For simplicity we assume that these conditions are always met automatically by the read-write head. The read-write head executes a POP operation by reading the topmost symbol and erasing it. A
PUSH operation is executed by simply writing symbols. This implementation corresponds to a typical array-based implementation of a stack, except that the array is assumed to be extendible to the right. To allow the automaton to check for an empty stack the first location is reserved for an initial pushdown symbol B, pronounced as 'bottom'. This symbol is never changed during a computation; when the pushdown only contains B it is said to be empty.

Second, the transition functions of pushdown automata are extended to allow them to depend on the topmost pushdown symbol.

A formal definition of the model:

A pushdown automaton $M$ is defined by the sextuple $(X,Y,Q,P,s,g)$, where

- $X$ is the set of inputs (the input alphabet);
- $Y$ is the set of outputs (the output alphabet);
- $Q$ is the finite set of states;
- $P$ is the set of pushdown symbols (the pushdown alphabet);
- $s$ is the transition function;
- $g$ is the start state.

![Diagram](image-url)

**Figure 2.3** Pushdown automata
The theoretical backgrounds

A pushdown automaton is initialized by

1. giving it an input word \( x \);
2. setting the read head over the first symbol of the input;
3. placing \( B \) as the only symbol in the pushdown;
4. setting the read-write head over \( B \);
5. setting the write head over the leftmost cell of the output tape;
6. setting the current state of the pushdown automaton to \( g \);
7. starting the pushdown automaton.

The basic execution cycle of the pushdown automaton is

1. read the symbol under the read head, that is the current symbol;
2. read the symbol under the read-write head, that is, the topmost pushdown symbol;
3. compute the next state, the output symbol and the replacement pushdown word from the current state, current symbol, and current pushdown symbol;
4. move the read head one cell to the right;
5. replace the topmost pushdown symbol with the computed pushdown word, resetting the read-write head to be over the new topmost pushdown symbol;
6. let the current state be the next state;
7. write the output symbol to the output tape;
8. move the writing head one cell to the right and the execution cycle has been completed.

These pushdown automata (Chomsky's type 2) are more general than the finite automata and they can be used for solving a larger class of problems.

The Turing machine (Chomsky's type 0) is the most general automaton. Turing machines are, however, much more than this. Originally introduced in the mid-thirties to provide a method for specifying algorithms, they have since then become the universally accepted basic computational model for algorithms, that is the touchstone. The Turing machine provides a theoretical model of computers which has withstood the test of time, almost 50 years in production and still going strong. It's the ultimate personal computer since only pencil and paper are needed, yet at the same time, it is as powerful as any real machine.

The Turing machine is, in many respects, similar to the finite automaton. It has a finite number of states and symbols and has transitions which depend on a symbol and state. But the similarity ends here since rewriting of the input symbols is permitted as is the writing of additional symbols. For this reason the Turing machine has a read-write head which is allowed a two-way movement.
To provide for the writing of additional symbols we assume that the Turing machine has an infinite tape or file of cells. Pictorially we have figure 2.4.

A Turing machine $M$ is specified by $M=(X,Y,Q,T,s,o,g,f)$, where

- $X$ is the set of inputs;
- $Y$ is the set of outputs;
- $Q$ finite set of states;
- $T$ an alphabet of tape symbols;
- $s$ transition function;
- $o$ output function;
- $g$ in $Q$ is a start state;
- $f$ in $Q$ is a final state.

A Turing machine is initialized by

1. resetting the read-write head over the first cell of the tape;
2. letting the state of the Turing machine be $s$;
3. placing an input word $x$ over $X$ into the first $|x|$ cells, one symbol to a cell;
4. assuming $B$ is in all other cells.

The basic execution cycle of the Turing machine then is

1. read the symbol under the read-write head, that is the current symbol;
2. compute the next state, the output symbol and the movement of the read-write head (Left, Right or None) word from the current state and the current symbol;
The theoretical backgrounds

(3) write the output symbol to the tape;
(4) let the current state be the next state;
(5) move the read-write head and the execute cycle has been completed.

Not only is the class of computable functions larger than those defined by finite and pushdown automata, but it includes all partial functions that can be computed by any formal machine, rewriting system, or programming language. For this reason it is hypothesized that whenever a new method of defining functions is introduced, these functions can always be computed by a Turing machine. This is known as Church's thesis, which implies that any effective computation, a mapping of strings of symbols to strings of symbols, can be carried out by a Turing machine. This thesis cannot be proved, since the notion of computability by a Turing machine is not a formalized notion, but an intuitive notion. However, evidence in its favor is overwhelming. Many methods of specifying computable functions have been introduced, for example, Post and Markov systems and multitape and multihead Turing machines, but in each case they have been shown to define exactly the Turing computable functions.

There is one class of abstract automata which has not yet been treated here: the linear bounded automata. This type of automaton (Chomsky's type 2) is less powerful (and general) than the Turing machine, but it is more powerful than the pushdown automaton. Effectively, a linear bounded automaton is a Turing machine in which the number of cells that can be used during a computation on an input word x is exactly |x| (the length of x).

For a more exact treatment of the theory of abstract automata we refer to [ARBIB], [HENNIE] and [WOOD]. In these books also information can be found about the next subject: the classical decomposition theory.

2.2 Classical decomposition theory

The decomposition of a complex system into simpler subsystems is a currently used technique, resulting from a series of observations. Among other points are the following:

(1) The system organization is made clearer: this is an important requirement for getting well-documented and easily maintainable systems.

(2) Every subsystem can be separately optimized, whereas it can be impossible directly to optimize the whole system.
Hence, the question naturally arises whether a given automaton can be decomposed into a series of smaller automata. The classical decomposition theory presents an answer to this question: yes, it is possible to decompose a given automaton, but the decomposed automaton is submissive to severe limitations. Only two, rather straightforward, decomposition methods can be applied. These methods, the parallel decomposition and the cascade decomposition respectively, will now be treated briefly.

Decomposed systems can be classified according to the number of submachines they contain and the manner in which these submachines are interconnected. As might be expected, the techniques used to find decomposed realizations depend on the method of interconnection.

One of the simplest ways in which a machine can be broken up into submachines is the parallel decomposition shown in figure 2.5. Here the submachines $M_1$ and $M_2$ are supplied with the same input sequence, but otherwise operate independently. Signals representing the internal states of $M_1$ and $M_2$ are supplied to a combinational circuit $C$ whose job it is to generate the output sequence. Parallel decomposition can be generalized in an obvious way to include the use of more than two submachines.

Another useful type of decomposition is the cascade decomposition shown in figure 2.6. Here again the overall realization has been broken up into two submachines, each driven by the same input sequence. But these submachines do not operate independently. Submachine $M_2$ is supplied, by means of auxiliary inputs, with information about the current internal state of submachine $M_1$. This information influences the state transitions that $M_2$ makes and
enables \( M_2 \) to generate the appropriate output sequence. The possibility of passing state information from \( M_1 \) to \( M_2 \) makes cascade decomposition somewhat more powerful than parallel decomposition. Indeed, parallel decomposition can be viewed as a special case of cascade decomposition.

Of course there are other ways in which machines can be decomposed. Figure 2.7 shows a type of decomposition in which each submachine is provided with information about the current state of the other. In this case the internal behavior of each submachine can be influenced by the behavior of the other, and neither one is completely controlled by the input sequence. Decompositions of this kind are with the given theoretical concepts usually difficult to find, and we do not treat them here.

In this paragraph we briefly recalled classical decomposition methods. These methods are based upon the theory of abstract automata and some concepts derived from the theory of discrete mathematics, e.g. covers, sets, lattices, groups etc. According to the general experience, those methods, when applied to practical sys
tems, lead to very long and tedious computations, except in rather trivial situations. It is for this reason that we let rest this subject and go ahead with other, more promising methods.

2.3 Concept of algorithm

In what follows next the concept of algorithm, as it is known in the theory of computation and algorithms ([SEDGEWICK] and [WOOD]), is presented.

According to this theory an algorithm is an abstract description of an information processing system. The information is, in order to be processed, appropriately coded into data. The algorithm states how, what and when data is to be processed. Essentially, an algo
Algorithm is no more than a sequence of instructions. These instructions specify actions. The actions are taking place in discrete steps. At each step there is an action, i.e. an event, which realizes a well defined effect. The action takes place in a certain environment, which is in a certain state. The effect of the action is a transformation of the state of the environment.

The environment in which the action takes place is characterized by a number of relevant quantities: the variables. The state of an environment is determined by a row of variables. A state transformation (caused by an action) is the changing of the value(s) of one or more variables.

An action can also be considered as consisting of a sequence of partial actions. An action which is constructed of a number of partial actions is called a process. According to this there exists an hierarchy of algorithms. An algorithm is a sequence of primitive actions and/or processes. A process also is a sequence of primitive actions and/or processes. So, from this it is clear that an algorithm is a sequence of primitive actions and/or algorithms (as a process is equivalent to an algorithm). In figure 2.8 this hierarchy of algorithms is visualized.

The sequence of actions in a process is controlled by acceptable control constructs. These constructs make the concept of algorithm more flexible: it is because of these constructs that it is possible to describe in a single algorithm a group of processes. Which process will take place is determined by the initial state of the system. To exemplify this, we now present a simple example:

Example 2.1

Consider the following PASCAL-algorithm:

```
ALGORITHM: example2.1;
BEGIN
  IF condition1 THEN
    BEGIN
      primitive_action1;
      primitive_action2;
    END
  ELSE
    IF condition2 THEN
      BEGIN
        primitive_action3;
        primitive_action4;
      END
  END;
```

The initial state of the system described by the algorithm depends on the information held by the variables, which span up the environment in which the actions take place. The values of the conditions `condition1` and `condition2` depend on the contents of the variables. If the initial contents of the variables, i.e. the initial state of the system, are such that `condition1` is TRUE and `condition2` is FALSE, the sequence of actions (the process) `primitive_action1`; `primitive_action2`; is specified by the algorithm. Otherwise, if `condition1` is FALSE and `condition2` is TRUE, the sequence of actions specified by the algorithm is `primitive_action3`; `primitive_action4`. Obviously, the initial state of the system determines which process of the two possible processes specified by the algorithm will take place.

In the conception of algorithms given above an algorithm is not a self-contained construct, but the presence of a second 'device', irrefrangible connected with the algorithm, is assumed: some computing agent capable of performing the primitive actions specified by the algorithm. In this view (figure 2.9), the algorithm determines the sequence of the primitive actions, by executing the acceptable control constructs, while the accessory computing agent executes these primitive actions.

![Algorithm Constructs](Algorithm)

![Computing Agent Actions](Computing Agent Actions)

**Figure 2.9 Algorithm and computing agent**

Like we stated in paragraph 2.1, a digital system is an information processing system. We can ask ourselves the question if it is possible to apply the theory of algorithms to digital systems. In order to be able to apply this theory to digital systems we must make sure that a digital system implements an algorithm (and, of course, its accessory computing agent).

From the theory of algorithms given above and our knowledge about digital systems we can extract the conditions which a digital system must satisfy in order to implement an algorithm (+computing agent).

The algorithm executes the acceptable control constructs and the computing agent executes the actions in discrete steps. The effect of the actions and constructs must be well-defined. This means that
Theoretical backgrounds

A digital system must execute the actions in a discrete, deterministic and unambiguous way.

As we saw in paragraph 2.1, a digital system is (in general) constructed of some combinational logic parts and some sequential logic parts, as shown in figure 2.10. We also saw in paragraph 2.1 that these combinational and sequential system parts, can be considered as finite automata.

Consider a digital system that, as we assume for the moment, implements an algorithm together with its accessory computing agent. Then the actions and the constructs specified by the algorithm must be executed by the digital system. The digital system is build up out of a number of sequential and combinational machines, and we adopt the following view: each sequential and combinational machine implements a primitive entity, i.e. a construct (in the case of the algorithm) or an action (in the case of the computing agent). These entities have to be executed by the machines in a discrete and deterministic manner. These constraints affect the possible realizations for the sequential and combinational machines, which we will study now.

The deterministic character of the machines could easily be jeopardized by the possible occurrence of non-deterministic effects like races, hazards and clock skewing. The occurrence of these undesirable effects can be prevented by adopting a proper set of design rules. We will now derive a set of rather strong rules.

We start off with the sequential machines. The discrete character of these machines can be easily obtained by implementing them as synchronous sequential machines. In this way, the complicated timing problems associated with asynchronous operation can be
Theoretical backgrounds

avoided. A sequential machine $M$ is said to be synchronous when it changes state only on the occurrence of a load signal at the storage devices. A general model for such a synchronous sequential machine is given below:

![Diagram of a synchronous sequential machine]

**Figure 2.11 A general synchronous sequential machine**

In the form presented so far, a synchronous sequential machine cannot function in a usable manner. Indeed, if $M$ is implemented with latches or edge-triggered flip-flops, abnormal behavior (critical races) can occur:

(a) if $M$ is implemented with latches, the transparent mode closes the circuit which then becomes an asynchronous automaton (with no storage devices). Its internal state can then change rapidly and unpredictably.

(b) if $M$ is implemented with edge-triggered flip-flops, the loading of the state by all elements of $M$ cannot occur simultaneously (clock skew). The rapid transition of some of the bits can disturb the present state and give rise to an unwanted transitory next state.

The solution generally adopted to resolve this problem involves the use of a two registers for $M$, $M$-master and $M$-slave, and the operation of the machine in two time slots called phases:

(a) an active phase in which the next state is calculated and stored in the $M$-master register while in transparent mode (or on receipt of a load transition). The $M$-slave register maintains stable data at the input to the combinational circuit;

(b) a passive phase in which information is transferred from the $M$-master register to the $M$-slave register.

23
If the storage elements are latches, then in order to avoid simultaneous transparent mode in both registers, which would cause unwanted asynchronous feedback around the combinational network, the two phases must not overlap.

The input values must be stable until the combinational logic has generated the transition function and the output function. To ensure this in a situation where several machines are interconnected, the inputs should be buffered. In this situation also a two-phase clocking scheme is necessary. The here developed model for a 'safe' sequential machine is given below:

![Diagram of a 'safe' sequential machine]

Figure 2.12 A 'safe' sequential machine

Of course, such a sequential circuit only functions well if the combinational propagation delay, together with the set-up time and propagation delay of the latches is smaller than the clock period. This means that the maximal combinational propagation delay is determined by the current clock period, or vice versa.

For the combinational machines now everything is simple: the inputs and outputs are, similar to the sequential machines, double-buffered. This prevents asynchronous operation and makes it possible to interconnect these machines with other machines (combinational and
Theoretical backgrounds

sequential) in a 'safe' way. The mentioned combinational machine is depicted in figure 2.12.

It is now clear that a digital system, serving the (strong) conditions mentioned, is nothing but a practical implementation of an algorithm. An algorithm is an abstract description of an information processing system, and a digital system of course is an information processing system. Furthermore, the natural hierarchy of algorithms is the basis for this report: the decomposition of digital systems supposes a hierarchy of digital systems.

![Figure 2.13 A 'safe' combinational machine](image)

Digital systems are finite automata and so, as shown in paragraph 2.1, both the theory of abstract automata and the theory of digital systems are derived from the same concept: the concept of algorithm. Now we have established the exact theoretical background of digital systems (figure 2.13).

The insight that a digital system implements an algorithm has been indicated by well-known authors like Davio [DAVIO1,2,3] and Glushkov [GLUSHKOV1,2].

**Remark:**

It is possible to formulate more specific and 'loose' conditions than the conditions derived in this paragraph. In [BUDZELAAR] a more profound treatment of this subject is given.
2.4 Correspondence with software design

A computer program, written in some computer language, together with the computer which executes the program, obviously form an implementation of an algorithm (with computing agent). The program specifies constructs and actions and both are executed by the computer. The program determines together with the sequencing mechanism of the computer the sequence of the actions, and thus forms this combination the algorithm. The computer executes the actions, and implements in that way together with the data upon which the actions take place, the computing agent. In figure 2.14 this scheme is illustrated.
Theoretical backgrounds

From this we see that also the design of software is based upon the concept of algorithm. The concept of algorithm links the hardware and software domains tightly together. This means that results obtained in the design of software systems can be used in the design of digital systems and vice versa.

The only difference between the process of digital system design and the process of software design is, that in the latter case the primitive actions, i.e. the instructionset of the computer or compiler, are known in advance. In the process of digital system design these primitive actions have to be chosen in some way or another. This extended degree of freedom makes digital system design more difficult than software design.

In the next chapter, an important decomposition model will be derived on the basis of the, in this chapter, presented theoretical backgrounds.
3. The development of the TUE decomposition model

In this chapter the so-called TUE decomposition model is derived on basis of a decomposition model proposed by Glushkov. In the next paragraph this model will be presented.

3.1 Glushkov's decomposition model

The famous Russian mathematician Glushkov proposed, in order to describe systems performing computations, a model of cooperation between two subsystems called operational automaton and control automaton respectively (Figure 3.1).

Consider the pair of automata A and B. Automaton A, which we will call the controlling one, serves to represent the algorithm, while automaton B, the so-called operational one, represents the information to be processed by the algorithm (with these objects being words, sets of words, numbers, etc.)

A transformation or an operator is what we call a mapping of the set M of these objects (the so-called information set) into itself. In addition to the operators on set M, conditions are defined which are true for some elements of M, false for others, and indeterminate for still a third group. Certain operators and conditions are elementary. The problem of the theory of algorithms consists in specifying methods of representing complex operators in terms of elementary operators and elementary conditions.
The operational automaton manipulates information and receives from the control automaton signals (commands) imposing the execution of actions such as register transfers. On the other hand, the operational automaton provides the control automaton with appropriate information about intermediate computation results. That information is provided under the form of condition variables.

Formally, the operational automaton may thus be described as the Moore automaton

$$B = (X, Y, Q_b, s_b, o_b)$$

where

1. the input alphabet $$X = (l, x_0, x_1, \ldots)$$ is the set of commands. It includes the empty command $$l$$ (no operation);
2. the output alphabet $$Y = (y_0, y_1, \ldots)$$ is the set of conditions;
3. $$Q_b$$, $$s_b$$, $$o_b$$ represent the state set, the transition function and the output function respectively. In particular, the transition function $$s_b$$ associates with each command $$x_i$$ a transformation $$(Q_b \rightarrow Q_b)$$ of the state set $$Q_b$$.

On the other hand, the control automaton is the actual implementation of the computation algorithm. It performs in the following way: taking into account the present computation step (its own internal state) and formerly obtained computation results (its inputs, the condition variables), it decides of the next computation step (next state) and of the action to be performed (output command). Formally, the control automaton will thus be defined as the Mealy automaton:

$$A = (Y, X, Q_a, s_a, o_a)$$

where the definitions are similar to those given for automaton $$B$$. Observe that the set $$Q_a$$ of internal states corresponds to the set of algorithm steps.

An algorithm is used to describe an information processing system. More specific, the algorithm specifies actions (or processes, being complex actions) which act upon the information (coded in data). The sequence in which the actions take place is specified by the control structures. The actions take place in an environment built out of the information upon which the actions work. This division of an information processing system into two subsystems, one representing the algorithm in which the constructs are executed, and the second one representing the information environment in which the actions are executed, is directly reflected in Glushkov's model:
In the algorithm concept it is assumed that all necessary information is kept in the operational automaton. So, no input or output operations are necessary. This suggests that, in the original concept, the operational automaton can best be modelled by an, in general, infinite Moore automaton. Indeed, the information that can be input into a system over a finite number of inputs is not limited to any bound.

Actual digital systems do perform in- and outputs and we feel the need to inflict these practical demands into Glushkov's model. This can be easily done by assuming that the operational automaton is not equal to the information environment, but implements just a part of it (figure 3.3). In this design, information can be input from the information environment to the operational automaton and, conversely, information from the operational automaton can be output to the information environment. Now, in this concept, the operational automaton can be modelled by a finite Moore machine.

In paragraph 2.3 some strong conditions were presented which made it possible to regard a digital system, which satisfied these conditions, as a practical implementation of an algorithm. From here on, we assume that every considered digital system will be realized in such a way that the alluded conditions are satisfied. In the rest of this report, we occupy ourselves with the decomposition problem in digital system design which, as we saw in the design cycle presented in chapter 1, is positioned between the formal specification phase and the implementation phase. Therefore, there is no need for us to consider the mentioned conditions any longer, as they are of sole importance to the realization phase. However, there is one condition that helps us to formulate the notions of operational automaton and information environment more precisely: the synchronous character of the system.
We attend ourselves to synchronized systems, in which some sort of a (central) synchronization mechanism (clock signal, load signal) is present. This synchronization mechanism takes care of the desired discrete character of the system at the basis: at each discrete time instant, e.g. t, the algorithm, i.e. the control automaton, initiates a primitive action by sending a command to the operational automaton. At the next primitive step of the algorithm, that is at time instant t+1, another primitive action is initiated. This second action could well be the same as the first action, which implies that it is necessary that a primitive action must finish its execution within the time period of the algorithm step. This can be guaranteed by assuming that all primitive actions eventually will be realized combinationally. The results of these combinational actions are available at the next step of the algorithm (the next clock cycle), provided the maximum combinational propagation delay and the current clock period do not form an impossible combination.

We now have the following situation: the operational automaton contains the information set M. This information set has a number of information elements (variables), for instance bits, bytes, words or numbers. These information elements are stored in storage elements, e.g. registers or flip-flops. Primitive actions are defined, which act upon the information elements in M. By means of primitive conditions, the relevant status of the information elements is signalled to the algorithm (the control automaton). The information environment now is formed by the information set M (in the operational automaton), and the environment of the operational automaton, from which information can be input to the M and vice versa.

We can now ask ourselves the question whether or not it is necessary to maintain the theoretically constructed notion of the
information environment. Digital systems perform in- and output operations. For this purpose in- and output terminals are available. Because the external information environment (not including the information set) contains all possible combinations of in- and output vectors, it is an infinite environment and as such it is part of another infinite environment: the natural environment of the system. For practical reasons it is not necessary to maintain the external information environment, and we can replace it with the natural environment, which is so obvious that it does not have to be explicitly present, or explicitly considered in Glushkov's extended decomposition model.

In the next paragraphs an attempt is made to clarify the in this paragraph presented notions by means of some examples.

**Remark:**

Glushkov's decomposition model is not so unknown as someone might think: the decomposition of a system in a datapath and a controller has been an accepted means for tackling the digital system design for a considerable length of time now. In this view, the datapath is equivalent to the operational automaton, and the controller is equivalent to the control automaton. We presented Glushkov's model apart from this the datapath/controller idea, because we would like to stress the importance of the theoretical basis of the decomposition model.

### 3.2 Sample decomposition 1: a simple transmitter

We want to test Glushkov's extended model by the following example: the system we want to decompose is a simple transmitter, which loads 8-bit words from the environment into the transmitter and transmits these words serially to the environment. The words are entered into the system via a 8-line databus. The serial bits are output to the environment by means of a single line, TxD. For the simplicity of the example we suppose that there is an infinite number of words to be transmitted (in other words, the environment contains an infinite number of words) and the transmitter performs the following task: First it loads a word from the environment, then it transmits this word, next it loads another word, it transmits it, etc. This sequence keeps going on forever, for we are dealing with an infinite number of words.

The outline of the transmitter is presented in figure 3.4.

We want to decompose this transmitter to the information set level, where combinational actions take place. By doing so, we shall try to respect an, as much as possible, hierarchical and top-down working method.
The behavior of the transmitter can be easily specified in an algorithm (Pascal-like notation):

```
ALGORITHM: transmitter;
BEGIN
  WHILE TRUE DO
    BEGIN
      load_word;
      transmit_word;
    END
  END;
```

In this algorithm we can distinguish the actions `load_word` and `transmit_word` and the condition `TRUE`, which is in fact no condition but a part of the while-loop, indicating that the loop must be repeated indefinitely. This algorithm implicates the following decomposition of the transmitter:
The TUE decomposition model

The control automaton executes the algorithm (control constructs) and initiates the actions load_word and transmit_word, which are executed in the operational automaton. The control automaton does not anticipate the sequence of its actions according to a certain condition or a certain set of conditions. Indeed, the control automaton is an autonomous automaton.

The specification method we have introduced here assumes that the control automaton has knowledge about the time span of the actions. This is necessary, because the control automaton has to start an action after the preceding action has finished. For instance, when the control automaton would start the action load_word before the action transmit_word has finished, the intended behavior wouldn't be guaranteed. In the preceding sections we assumed in silence that the control automaton did indeed have knowledge about the duration of the actions, and was capable of initiating the actions in the specified and intended sequence.

However, in the case of digital systems, at a high level, were we do not know yet whether or not an action is complex, or how long it takes for a complex action to finish its execution, it is necessary to provide some means of synchronization in order to keep up the sequential coherence of the actions.

For that purpose we introduce in the example of the transmitter the synchronization signals transmit_word_ready and load_word_ready. Transmit_word_ready indicates that the action transmit_word is not active and load_word_ready of course indicates that the action load_word is inactive. The behavior of the transmitter can now be described by:

```
ALGORITHM: synchronized_transmitter(1);

BEGIN
WHILE TRUE DO
BEGIN
  WHILE NOT(transmit_word_ready) DO nothing;
  load_word;
  WHILE NOT(load_word_ready) DO nothing;
  transmit_word;
END
END;
```

The action nothing represents the empty command l. nothing is no action in the usual sense, but its appearance just indicates that no action is active (all command signals, i.e. load_word and transmit_word etc., are inactive).

This algorithm also implicates a certain decomposition of the transmitter (figure 3.6): The control automaton has as its inputs the synchronization signals transmit_word_ready and load_word_ready,
and the actions load_word and transmit_word as its outputs. The operational automaton executes these actions and provides the control automaton with the synchronization signals.

When one of the two actions is being executed, the corresponding synchronization signal is made FALSE, so that the other action cannot be started. When the execution of the action finishes, the synchronization signal becomes TRUE and the next action can be initiated by the control automaton.

![Diagram of synchronized transmitter](image)

**Figure 3.6 The synchronized transmitter**

Apparently there are two kinds of conditions:

- status conditions, representing relevant status information about the processes in the system;

and

- synchronization conditions, representing the process information necessary for the algorithm to initiate the actions at the right time.

On the basis of the status conditions, the algorithm decides which action to initiate next. Status conditions affect the action sequence; they determine which process is executed by the algorithm. Synchronization conditions however, do not affect the action sequence, they only guard the intended (sequential) coherency of the algorithm.
The TUE decomposition model

We now return to our example: we want to decompose the transmitter to the information set level (combinational actions). Therefore we must decompose the complex actions and conditions (in this case only synchronization signals) into primitive actions and conditions. Consider the action load_word: is this a complex action or a primitive action? Let's study the element of the information set upon which this action takes place:

The action load_word acts upon the information set element '8-bit word', which we intuitively feel can be represented by an 8-bit register. When this register is built out of flip-flops which have a load control input, we see that the action load_word is equal to the primitive action load_register. Thus, the action load_word is primitive and can easily be implemented as shown in figure 3.7. Because this action is primitive, the execution of the action has finished within the primitive time step (algorithm time step, clock cycle) and the synchronization signal load_word_ready appears to be superfluous.

![Figure 3.7 Implementation of load_word](image)

The transmitter now can be described by the algorithm:

```
ALGORITHM: synchronized_transmitter(2);
BEGIN
WHILE TRUE DO
BEGIN
    WHILE NOT(transmit_word_ready) DO nothing;
    load_word;
    transmit_word;
END
END;
```
We can adopt this method also to the other action, transmit_word: because this action has to perform a conversion from the 'word-level' (register) to the 'bit-level' (serial output TxD), and considering the fact that the 8 bits of a dataword have to be transmitted over TxD one after the other, we feel that transmit_word is a complex action that still can be decomposed. A very simple and straightforward approach is represented by the next algorithm:

**ALGORITHM: transmit_word(1);**

BEGIN
transmit_word_ready:=FALSE;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
transmit_word_ready:=TRUE;
END;

The algorithm initiates 8 times the action transmit_bit in a row. Each transmit_bit action performs the transmission of one bit of the word held in the register. Because we don't know yet whether or not transmit_bit is a primitive action, we have provided the synchronization signal transmit_bit_ready.

![Figure 3.8 Synchronized_transmitter(3)](image)
What we have done here is, in fact, the expansion of the complex action `transmit_word` into a sequence of other (simpler) actions. The complete transmitter (figure 3.8) is now described by:

```
ALGORITHM: synchronized_transmitter(3);
BEGIN
WHILE TRUE DO
BEGIN
WHILE NOT(transmit_word_ready) DO nothing;
load_word;
transmit_word_ready:=FALSE;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
WHILE NOT(transmit_bit_ready) DO nothing;
transmit_bit;
transmit_word_ready:=TRUE;
transmit_word_ready:=TRUE;
END
END;
```

The control automaton is hierarchically constructed. The transmitter algorithm functions as the main controller which initiates primitive actions (`load_word`) and complex actions (`transmit_word`). The complex action `transmit_word` can be considered as a subcontroller, which initiates the primitive action `transmit_bit` to the operational automaton. The assignment actions `transmit_word_ready:=FALSE` and `transmit_word_ready:=TRUE` are not common actions, in the sense that they are initiated by the subcontroller `transmit_word` and executed by the operational automaton (generally together with some subcontrollers), but they are initiated and executed within the subcontroller `transmit_word`. They serve the following purpose: the synchronization signal `transmit_word_ready` (the value of which is changed by these synchronization actions) is used by the main controller to maintain the intended coherence of the transmitter algorithm. In figure 3.9 the control hierarchy of `synchronized_transmitter(3)` is shown.
The notation of algorithm transmit_word(l) is very tedious and can be simplified as follows:

```
ALGORITHM: transmit_word(2);
BEGIN
    transmit_word_ready:=FALSE;
    FOR i:=1 TO 8 DO
        BEGIN
            WHILE NOT(transmit_bit_ready) DO nothing;
            transmit_bit;
        END
    transmit_word_ready:=TRUE;
END;
```

In this case we have introduced an auxiliary variable i, with which it is possible to formulate the transmit_word algorithm in a more brief way. This algorithm initiates the actions exactly in the same sequence as the first transmit_word algorithm, so these algorithms are equivalent to each other. The variable i does not have any
effect on how the actions are executed. Thus, it belongs purely to the \texttt{transmit\_word}(2) algorithm and it is a variable that is maintained within the subcontroller \texttt{transmit\_word}.

The behavior of the transmitter can now described by the following algorithm:

\begin{verbatim}
ALGORITHM: synchronized\_transmitter(4);
BEGIN
WHILE TRUE DO
BEGIN
WHILE NOT(transmit\_word\_ready) DO nothing;
load\_word;
transmit\_word\_ready:=FALSE;
FOR i:=1 TO 8 DO
BEGIN
WHILE NOT(transmit\_bit\_ready) DO nothing;
transmit\_bit;
END
transmit\_word\_ready:=TRUE;
END
END;
\end{verbatim}

This algorithm is equivalent to the \texttt{synchronized\_transmitter}(3) algorithm. Ergo, we have pictorially the same figures (figure 3.8 and figure 3.9).

There is still another way in which the complex action \texttt{transmit\_word} can be decomposed: we can make use of a counter which keeps count of the number of bits already transmitted. First the counter is reset, and then after each transmission of a bit (after each execution of the action \texttt{transmit\_bit}) the value of the counter is incremented. Before another bit is transmitted the value of the counter is compared to the number 8. When the value of the counter equals 8, it is known that the entire word is transmitted and the control can be returned to the main controller. The corresponding algorithm is shown beneath:

\begin{verbatim}
ALGORITHM: transmit\_word(3);
BEGIN
transmit\_word\_ready:=FALSE;
count:=0;
REPEAT
BEGIN
WHILE NOT(transmit\_bit\_ready) DO nothing;
transmit\_bit;
count:=count+1;
END
UNTIL count=8
transmit\_word\_ready:=TRUE;
END;
\end{verbatim}
Now we have the actions \( \text{count}=0 \), transmit\_bit and \( \text{count} = \text{count}+1 \), and the conditions transmit\_bit\_ready (synchronization condition) and \( \text{count}=8 \) (status condition).

We have excluded the synchronization signals associated with the actions \( \text{count}=0 \) and \( \text{count} = \text{count}+1 \) because it is immediately clear that these actions can be implemented combinationally. The variable \( \text{count} \) (value range 0 - 8) can be represented by a 4-bit counter, which has a reset or clear input (\( \text{count}=0 \)) and a count or increment input (\( \text{count} = \text{count}+1 \)), as shown in figure 3.10. In this figure also the implementation for transmit\_bit is given: the register that we established earlier is upgraded to a shift-register. Besides a load input it has now also a shift input which is directly connected to the transmit\_bit command line. The serial output data line TxD is connected to the right most bit cell of the shift register. Every time a transmit\_bit command is given, the contents of the shift-register is shifted one bit cell to the right.

![Diagram](image)

**Figure 3.10 Expansion of the transmitter with an 8-bit counter**

Because the transmit\_bit action appears to be a primitive action, the transmit\_bit\_ready synchronization signal can be omitted: it is now guaranteed that the execution of the action has finished at the beginning of the next step of the algorithm.

The condition \( \text{count}=8 \) places us for a problem: is this condition evaluated at the time we need it, that is at the next step of the algorithm. Immediately after the value of count is updated (\( \text{count} = \text{count}+1 \)), the value of the condition is needed in order for
the algorithm to decide whether or not the control can be returned to the main controller. What if this is not the case? Then it is possible that a ninth bit (or even more) is transmitted, because the condition $\text{count}=8$ didn't give the value TRUE when it was needed. The transmitter doesn't operate correctly.

What can we do to prevent this? First, it is possible to make use of a synchronization signal associated with the condition, in the same way as this was the case for the complex actions. Secondly, it is possible to expand the complex condition to a number of actions and a primitive condition. In our case we can expand the condition $\text{count}=8$ to the following sequence:

```
reset_comparator;
REPEAT
    .
    .
    .
    compare(count,8);
UNTIL compare_flag
```

We imagine a comparator capable of comparing 4-bit values ($1000_2=8$). Before a word is transmitted, the comparator is reset and the `compare_flag` becomes FALSE. Each time a bit is transmitted, the comparator compares the value of `count` with the number 8, and when these values match the `compare_flag` is made TRUE. Of course, this decomposition of the (appearingly complex) condition $\text{count}=8$ wouldn't be of any use if the condition `compare_flag` also appeared to be complex. Well, this doesn't have to be the case: the comparator, the operations `reset_comparator` and `compare(count,8)` and the condition `compare_flag` can be conveniently combinationally implemented at the information set level, as shown in figure 3.11.

There is still a third possibility to implement $\text{count}=8$: the value of `count` can be led to the subcontroller `transmit_word`, where it can be compared to the number 8. This, of course is a nasty implementation method because we metaphorically use the variable `count` completely as a condition, and that is not what we intended at the beginning. We decided to draw a line between the information upon which the actions works, and the algorithm, specifying the sequence of the actions. When the complete values of variables are transmitted to the algorithm, i.e. to the control hierarchy, this separation is affected. Therefore, we drop this last possibility and limit ourselves to implementations obeying the boundary between algorithm and information.
When we adopt transmit_word(3) the algorithm for the transmitter becomes:

**ALGORITHM: synchronized_transmitter(5);**

**BEGIN**

**WHILE TRUE DO**

**BEGIN**

**WHILE NOT(transmit_word_ready) DO nothing;**

**load_word;**

**transmit_word_ready:=FALSE;**

**count:=0;**

**reset_comparator**

**REPEAT**

**BEGIN**

**transmit_bit;**

**count:=count+1;**

**compare(count,8);**

**END**

**UNTIL compare_flag**

**transmit_word_ready:=TRUE;**

**END**

**END;**

In order to be complete the *synchronized_transmitter(5)* algorithm is visually reproduced in figure 3.12.

---

*Figure 3.11 Expansion with a 4-bit comparator*
3.3 Consequences for Glushkov's model (1)

From the simple transmitter example we can derive some general notions about the decomposition problem. First of all, it is clear that we can represent a general decomposed system by the model presented in figure 3.13.

As we saw in paragraph 3.2, the control automaton is a hierarchical construction, build up from a single main controller (control level 0) and a number of subcontrollers (situated at the control levels 0,...,N). Each subcontroller only communicates with it's master (sub)controller (the (sub)controller directly above in the hierarchy) and with all subcontrollers directly beneath in the control hierarchy ('slave subcontrollers'). Communication between the (sub)controllers is split up in three different signal types.
Figure 3.13 A general decomposed system
The TUE decomposition model

(1) **commands:**

A (sub)controller (master) can initiate actions in its slave-controllers, by sending commands to them. With every command there is precisely one single slave-controller associated;

(2) **status conditions:**

A slave-controller can signal intermediate results (status) to its master-controller, which can make use of this information to anticipate and control the initiation of actions in the slave-controllers. With every master-slave relation, there are generally several status conditions associated;

(3) **synchronization conditions:**

These signals serve the purpose of maintaining the coherence of the master-control algorithm. They enable the master-controller to maintain the intended action sequence of the master-control algorithm. With every action (or command), and thus with every subcontroller, there is at most one synchronization condition associated.

This last remark, that for every subcontroller at most one synchronization signal will do, is only true in the case that all conditions are primitive conditions, i.e. booleans. This means that all conditions, as was the case in the example, are decomposed into a number of actions (local to the subcontroller) and a single boolean condition (value: TRUE or FALSE) upon which the master-controller can take the desired action. This procedure is very similar to software design, were every condition first is evaluated (decomposed into a number of actions) and finally is reduced to a boolean condition (primitive condition). It makes certain that the condition is ready at the next step of the algorithm (when it is needed). Therefore no synchronization signals associated with these primitive conditions are necessary.

In the general model the distinction between the control automaton and the operational automaton is very clear. The control automaton implements the algorithm, and the operational automaton implements the information set. At this level, only primitive (combinational) actions and conditions are defined. The result of every action is present at the next step of its master-controller algorithm, so that no synchronization signals at all are needed. However, we also saw that the boundary between information set and algorithm is not so 'solid' as we might have hoped for. In the example this comes forward in the case of the transmit_word algorithm: there it appeared to be possible to implement the mechanism that keeps count of the number of already transmitted bits in at least two ways. In the first method (transmit_word(2)) the state variable belonging to the subcontroller transmit_word implemented the count mechanism. In
this case, the count mechanism thus belongs to the algorithm (=control automaton). In the second case (transmit_word(3)), the count mechanism was explicitly implemented at the information set level, and thus belonged to the information set (=operational automaton). Apparently, the implementation of the system and the associated boundary between the control automaton and the operational automaton, strongly depends on decisions made at the algorithm level.

Unfortunately, there are at this moment no methods available by which it is possible to decide at a high level what decisions result in the best implementation. Therefore, a pure top-down design method cannot be used. Wherever in the design cycle a decision has to be made of which the consequences for the implementation are not clear or straightforward, it might be necessary later on in the design process to return to that decision point, and to consider a different decision. In the rest of this report this approach will be indicated by the term 'yo-yo'.

In the general decomposition model (figure 3.13) this design approach also comes forward. At a high level we generally cannot decide if a particular action is primitive or complex. Therefore, in order to leave all possibilities open, we have to assume that the action is complex and that a subcontroller together with a synchronization condition is necessary to implement this action. Later on it might become clear that the mentioned action is primitive and, as a consequence, the subcontroller and the synchronization condition can be replaced by a simple signal line. The decomposition model thus represents the most general case, and in practical situations it is possible that a subcontroller represents a signal line. See, for instance, figure 3.12.

As far as the decomposition problem is concerned, we can distinguish two important steps: the first one, is the top-down decomposition, resulting in the general decomposition model, as indicated in figure 3.13. The second step is the bottom-up refinement, in which the superfluous control structures can be replaced with simple signal lines.

3.4 Sample decomposition 2: a CPU-controlled transmitter

Let's apply the method derived in the previous example to a new problem: the decomposition of a transmitter that is (partially) controlled by a central processing unit (figure 3.14). The CPU can initiate the transmission of a dataword by writing an 8-bit word over the databus to the transmitter. This operation is controlled by the write signal. When the transmitter is idle (has no word to transmit) the line transmitter_ready signals to the CPU that it is possible to write a new dataword to the transmitter.
The TUE decomposition model

Figure 3.14 A CPU-controlled transmitter

Analogous to the previous example the algorithm for the CPU-controlled transmitter can be written down as:

ALGORITHM: CPU-controlled_transmitter;
BEGIN
WHILE TRUE DO
BEGIN
WHILE NOT(transmit_word_ready AND write) DO
nothing;
load_word;
transmitter_ready:=FALSE;
transmit_word;
transmitter_ready:=TRUE;
END
END;

Here we have taken advantage of the knowledge gathered in the last example. There it was established that the action load_word can be implemented as a primitive action, and therefore no synchronization signal load_word_ready is necessary.
The TUE decomposition model

The actions transmitter_ready:=FALSE and transmitter_ready:=TRUE are used to control the synchronization signal transmitter_ready, which is output to the CPU. With these actions no synchronization signals are needed, because we can easily imagine an information set level implementation: the command signals transmitter_ready:=TRUE and transmitter_ready:=FALSE can be used to control the set and reset inputs of a set-reset flip-flop respectively. The output of the flip-flop is directly connected to the transmitter_ready output.

The actions appear to be primitive, and therefore we can do without synchronization signals.

At this moment a strong suspicion arises that the actions transmitter_ready:=TRUE and transmitter_ready:=FALSE can be omitted from the algorithm, because we suspect that they are equivalent to the actions transmit_word_ready:=TRUE and transmit_word_ready:=FALSE which, as we did see in the last example, are used in the algorithm which implements transmit_word. As we proceed we will keep this suspicion in mind.

The write signal is an interesting signal: from the view of the CPU it acts as a command signal, telling the transmitter that another word can be loaded and transmitted. From the view of the transmitter, and especially from the view of the main controller (control level 0), this signal behaves as a synchronization condition, indicating to the main controller that the CPU is ready with its preparations: a dataword can be loaded and transmitted.

For the moment, we conform ourselves to the latter view, and regard the write signal as being a synchronization condition. This means that this signal has to be supplied to the main controller from a lower level of the transmitter's hierarchy (we saw: conditions are communicated from one level to a higher level of the hierarchy). This demand corresponds to Glushkov's decomposition model, in which in- and output operations are performed at the lowest level: the information set level. It is here that the write signal enters the transmitter, from where it is directly put through to the main controller.

A first decomposition of the CPU-controlled transmitter in a control automaton and an operational automaton according to Glushkov's model is given in figure 3.15.

When we recall the requirements which a decomposition method should come up to (see chapter 1), the demand that such a decomposition method should proceed in a, as much as possible, hierarchical manner, was very strong. In the underlying example we want to conform ourselves to this demand and decompose the system hierarchically.
The TUE decomposition model

Consider the in figure 3.15 depicted (obvious) decomposition of the system in a control automaton and an operational automaton.

When we want to decompose this system hierarchically, we have to decompose each subsystem, i.e. the control automaton and the operational automaton, apart from each other. This means that the control automaton implements only the main controller algorithm, and thus that the operational automaton implements the rest of the system, i.e. the lower control levels and the information set level. In figure 3.16 a model representing the here derived boundary between control automaton and operational automaton is presented.

Let us now return to the example of the CPU-controlled transmitter. In order to decompose the system further, we have to elaborate the complex action transmit_word. For this purpose any of the transmit_word algorithms given in paragraph 3.2 can be used. We make an (arbitrary) choice and pick the transmit_word(2) algorithm, which is reproduced here:

**ALGORITHM: transmit_word(2);**

**BEGIN**

transmit_word_ready := FALSE;
FOR i:=1 TO 8 DO
  transmit_bit;
transmit_word_ready := TRUE;
**END;**

Like we said before, the actions transmit_word_ready := TRUE and transmit_word_ready := FALSE are local to the subcontroller which implements the transmit_word(2) algorithm.

The in this way decomposed system is given in figure 3.16.

---

**Figure 3.15 A first decomposition**

![Diagram of the control automaton (CA) and operational automaton (OA) with connections and actions between them.]

- **write**
- **transmit_word_ready**
- **transmitter_ready := FALSE**
- **transmitter_ready := TRUE**
- **load_word**
- **transmit_word**
- **TxD**
- **transmitter_ready**

The diagram illustrates the connections and states between the control and operational automata.
Figure 3.16 The hierarchically decomposed transmitter
The TUE decomposition model

The algorithm for the total system now is:

ALGORITHM: CPU-controlled transmitter;

BEGIN
WHILE TRUE DO
BEGIN
WHILE NOT(transmit_word_ready AND write) DO
nothing;
load_word;
transmitter_ready:=FALSE;
transmit_word_ready:=FALSE;
FOR i:=1 TO 8 DO
transmit_bit;
transmit_word_ready:=TRUE;
transmitter_ready:=TRUE;
END
END;

It is immediately clear that our suspicion was founded: the actions transmitter_ready:=TRUE and transmitter_ready:=FALSE are superfluous. The actions transmit_word_ready:=TRUE and transmit_word_ready:=FALSE are equivalent to them and the synchronization signal transmit_word_ready can be used to signal transmitter_ready to the CPU.

As we can see in figure 3.16 the transmit_word_ready signal is both available in the operational automaton (subcontroller transmit_word) and the control automaton. Because Glushkov's model only permits in- and output operations at the information set level, the transmit_word_ready signal has be output to the CPU via the information set level. The artificiality, that emerged in the elaboration of this example, gives rise to the question whether or not it is wise to hold on to Glushkov's model, and the limitations associated with it, or to extend the model so that it can be used elegantly to cope with practical situations.

Of course, we choose for the last possibility. In order to set up a new model we first have to study the awkward limitations we encountered in this example.

First, there is the write signal. This signal is only needed at the highest level (main controller), but it is input at the lowest level (information set). The signal is unchangeable transmitted from the information set level to the main controller. This unnecessary and confusing detour can be avoided by permitting appropriate (in our case: the write signal) signals to be input to the main controller.
Originally we assumed that the write signal was a synchronization condition. Now we can no longer persist in this view. We did see that (synchronization) conditions are only communicated from a controller to a controller at a higher level. This means that the CPU can be considered as a lower level controller, opposite to the main controller. Intuitively we feel that this is not correct: the CPU controls the transmitter. Therefore, we can consider the CPU as being at a higher level than the main controller. A higher control level controls a lower level by means of command signals and associated conditions. From this it immediately follows that the write signal is a command signal, and that the transmitter_ready signal is its associated synchronization signal (Figure 3.17). To protect the coherence of the algorithm, a synchronization signal should be issued from the same control level as where the action (as result of the command signal) is executed. Thus, the transmitter_ready signal must be generated at the same level as where the write signal enters, e.g. the main control level. This new insight is visually represented in Figure 3.18.

![Diagram](image)

**Figure 3.17 The CPU controls the transmitter**
3.5 Consequences for Glushkov's model (2)

In this chapter we worked out two examples, and it is on the observations we constantly made that we came to the following conclusions:

(1) To inflict practical demands, Glushkov's model was enhanced in such a way that it now is possible for the control automaton to communicate directly with the outside;

(2) We saw two possible elaborations of Glushkov's model: the first (figure 3.13) was encountered in example 1 and will be referred
to as model 1, the second (figure 3.19) was encountered in example 2 and will be referred to as model 2.

The boundaries between operational automaton and control automaton associated with each model are quite different: in model 1 the complete algorithm (main controller + all subcontrollers) is situated in the control automaton. The operational automaton only implements the information set level (storage of variables and constants, primitive actions and conditions). This separation completely respects Glushkov's model and the concept of algorithm: the control automaton implements the algorithm and the operational automaton functions as the associated computing agent.

Unfortunately, the decomposition method leading to model 1 does not satisfy an important requirement: it is not purely hierarchical. The first decomposition in a control automaton and a operational automaton is not respected throughout the decomposition process.

The decomposition method leading to model 2, in which the control automaton only implements the main control level and the operational automaton the rest, is purely hierarchical. However, a point of consideration must be that this model does not respect the boundary presented by Glushkov's model and the concept of algorithm: the algorithm is split up between the control automaton and the operational automaton.

At this point we have to make a choice between the two models, because we want to examine the decomposition problem further, and in more detail. We choose for the second model, despite of the fact that this model does not respect the boundary between the control automaton and the operational automaton presented by Glushkov's model and the concept of algorithm. This decision has been made on the basis of the following considerations:

(a) Glushkov's model was not a goal on itself, but just a means by which a hierarchical decomposition method can be found;

(b) The character of the main controller is much more that of an independent controller than that of the other controllers. The subcontrollers have a more performing and dependent character: they execute what the main controller tells them to do. If we adopt this view, we could say that the subcontrollers intrinsically belong to the computing agent, i.e. the operational automaton. This separation between a controlling and independent part (the main controller) on the one side, and a performing and dependent part (the lower control levels + the information set level) on the other side is more or less a reflection of the separation between the algorithm and the computing agent, as was presented by the concept of algorithm and Glushkov's model. Indeed, the algorithm controls the computing agent and has a independent character, whereas the computing agent executes the commands given by the algorithm, and presents a dependent character. Considering this, we could say
The TUE decomposition model

Figure 3.19 Decomposition model 2
The TUE decomposition model

that model 2 respects, more or less, the concept of algorithm, i.e. the boundary between algorithm and computing agent;
(c) The hierarchical nature of model 2.

Summarizing the accumulated knowledge about the decomposition problem thus far, we can say that we have developed a general decomposition model (model 2) and an associated decomposition method with which it is possible to decompose a system in a hierarchical and nearly top-down procedure. In the next chapter we will add an important feature to this: parallelism.
4. Parallelism

4.1 Introduction

Until now, we only studied sequential algorithms and implementations. In the preceding sections, Glushkov’s decomposition model appeared as a paragon of sequentialism: at a given (control) level, at most one command at a time is active. Therefore, the concurrent execution of two actions cannot occur.

In chapter 1, it was stated that the decomposition method we intended to find should inhibit some qualities: it should be hierarchical, and it should result in an as much as possible, optimal decomposed system. Of course, optimal is a very subjective term. It indicates that the system should fulfil some prior defined constraints, generally related to the operating speed of the system or to the number of gates necessary to implement the system. This last feature is usually denoted as the cost of the system. It is here that we enter the extensive and complex field of the trade-off problems: generally, the speed of a given system is proportional to the cost of that system, and therefore it is very difficult to fulfil at the same time high speed and low cost constraints. The best thing to do in such a situation is to design the system in such a way that the most important constraint is fulfilled, and the other one as good as possible.

All the decomposition methods we saw before resulted in a purely sequential system. This means that the resulting systems were relatively slow and expensive. In order to be capable of designing systems with a higher speed the design and decomposition methods must be extended to incorporate parallelism: by executing several commands at the same time the throughput of the system can be increased.

In this chapter we are going to investigate the effects of the introduction of parallelism to the decomposition problem in digital system design. Before we go deeper into the theory of parallel processes, we first present two simple problems, which help us to better understand some notions of the problematic nature involved.

4.1.1 Sample decomposition 3: transmitter with parity

We return now to our well-known autonomous transmitter (see example 1). We want to extend this transmitter with a parity generation mechanism: before a word is transmitted, the (even) parity of the eight bits is calculated and the resulting parity-bit is appended to the dataword. At the end of the transmission of the eight databits, the parity-bit is also transmitted: the transmission of a word involves the transmission of nine successive bits.
The following sequential algorithm can immediately be derived:

\[\text{ALGORITHM: transmitter\_with\_parity;}
\]

\[\text{BEGIN}
\text{WHILE TRUE DO}
\text{BEGIN}
\text{WHILE NOT(transmit\_word\_ready) DO nothing;}
\text{load\_word;}
\text{generate\_parity;}
\text{WHILE NOT(generate\_parity\_ready) DO nothing;}
\text{transmit\_word;}
\text{END}
\text{END;}
\]

\[\text{Figure 4.1 Implementation of the transmitter with parity}
\]

The complex action generate\_parity can be decomposed into two (primitive) actions:

\[\text{ALGORITHM: generate\_parity;}
\]

\[\text{BEGIN}
\text{calculate\_parity;}
\text{load\_paritybit;}
\text{END;}
\]

The parity of the dataword is calculated in the parity generator, a combinational circuit constructed from a number of exclusive-or gates. The shift-register, which is used to store and transmit the dataword, is extended with a ninth flip-flop, which is used to store the parity-bit. In figure 4.1 the transmitter circuit is given.
The number of bits to transmit has gone up from eight to nine, meaning that the algorithm which implements `transmit_word` has to change a little:

```
ALGORITHM: transmit_word;
BEGIN
    transmit_word_ready:=FALSE;
    FOR i=1 to 9 DO
        transmit_bit;
        transmit_word_ready:=TRUE;
    END;
END;
```

Finally, the here established control hierarchy of the (sequential) transmitter is shown in figure 4.2.

![Control hierarchy of the transmitter with parity](image)

**Figure 4.2 Control hierarchy of the transmitter with parity**

At the main control level, the ordering of the actions initiated can be expressed in a so-called time trace:
The execution period of each action is expressed in (relative) time units or clock periods.

The action `load_word` is primitive and needs only a single time unit for its execution. `Generate_parity` is a complex action which inhibits the execution of two primitive actions (two time units). Finally, `transmit_word` is constructed of nine consecutive primitive operations (nine time units).

The time needed to transmit a single word can be used as an expression for the relative speed of the sequential transmitter. This time is equal to `load_word (1) + generate_parity (2) + transmit_word (9) = 12` time units.

When we examine the time trace of the `transmitter_with_parity` in more detail, it becomes clear that, with the given implementation (figure 4.1), it is not possible to execute any of the actions `load_word`, `generate_parity` and `transmit_word` in parallel:

- `load_word` and `transmit_word` use both the same shift-register;
- `transmit_word` and `generate_parity` use both the ninth bit (parity-bit) of the shift-register;

The nature of the mentioned pairs of actions is so different (they use the common resources in a mutual exclusive manner), that concurrent execution is out of the question. However, the actions `load_word` and `generate_parity` could be executed in parallel (they don't use the same resource), but unfortunately, the next to be transmitted dataword would be lost.

In order to make the parallel execution of `load_word` and `generate_parity` possible and thereby speeding up the system, we make use of a well-known technique called pipelining. This technique involves the introduction of extra memory elements, which make it possible to execute actions, which use these memory elements as resources, in parallel.

In our case, we can speed up the transmitter by adding a pipeline register (buffer) to the circuit as depicted in figure 4.3. This buffer register can be loaded from the databus by the primitive command `load_buffer`. All other commands have remained the same, although
we have renamed load_word to load_register for clarity reasons.

Figure 4.3 Extension with a pipeline register

This circuit can be described by the following (still sequential) algorithm:

ALGORITHM: pipelined_transmitter(1);
BEGIN
WHILE TRUE DO
BEGIN
WHILE NOT(transmit_word_ready) DO nothing;
load_buffer;
generate_parity;
WHILE NOT(generate_parity_ready) DO nothing;
load_register;
transmit_word;
END
END;

The accessory control hierarchy is given in figure 4.4. Again, it is possible to express the ordering of the actions at the main control level in a time trace:
Parallelism

<table>
<thead>
<tr>
<th>relative time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load_buffer</td>
</tr>
<tr>
<td>1</td>
<td>generate_parity</td>
</tr>
<tr>
<td>3</td>
<td>load_register</td>
</tr>
<tr>
<td>4</td>
<td>transmit_word</td>
</tr>
<tr>
<td>13</td>
<td>load_buffer</td>
</tr>
<tr>
<td>14</td>
<td>generate_parity</td>
</tr>
<tr>
<td>16</td>
<td>load_register</td>
</tr>
<tr>
<td>17</td>
<td>transmit_word</td>
</tr>
<tr>
<td>26</td>
<td>load_buffer</td>
</tr>
</tbody>
</table>

etc.

The time needed to transmit a single word is here equal to
load_buffer (1) + generate_parity (2) + load_register (1) +
transmit_word (9) = 13 time units. This period is longer than the
not-pipelined transmitter, but we have not yet been using the
intrinsic parallelism of the system.

Figure 4.4 Control hierarchy of the pipelined transmitter

By examining the time trace thoroughly, it becomes clear that the
following pairs of actions can be executed concurrently:
Parallelism

The symbol \( \parallel \) indicates the concurrent execution of the actions.

Of course, the concurrent execution of these actions can only proceed in a correct manner if some proper conditions are met. For instance, in order to correctly execute the actions \texttt{load_buffer} and \texttt{generate_parity} in parallel, the following situation must exist prior to the execution:

(1) the shift-register must be loaded with the present contents of the buffer register;
(2) \texttt{transmit_word_ready} must be active.

From the above mentioned pairs of actions we use \texttt{load_register \parallel generate_parity} and \texttt{load_buffer \parallel transmit_word} to describe the transmitter:

\[
\text{ALGORITHM: pipelined_transmitter(2);}
\]

\[
\text{BEGIN}
\text{WHILE TRUE DO}
\text{load_buffer;}
\text{BEGIN}
\text{WHILE NOT(transmit_word_ready) DO nothing;}
\text{load_register \parallel generate_parity;}
\text{WHILE NOT(generate_parity_ready) DO nothing;}
\text{load_buffer \parallel transmit_word;}
\text{END}
\text{END;}
\]

The synchronization conditions are easily determined: just take, of a number of parallel actions, that action that has the longest execution period. If this cannot be determined in advance, the two synchronization signals have to be combined into one. When the execution of this action has finished, control can be returned to the main controller.

The time trace of the actions at the main control level is:

<table>
<thead>
<tr>
<th>relative time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{load_buffer}</td>
</tr>
<tr>
<td>1</td>
<td>\texttt{load_register \parallel generate_parity}</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{load_buffer \parallel transmit_word}</td>
</tr>
<tr>
<td>12</td>
<td>\texttt{load_register \parallel generate_parity}</td>
</tr>
<tr>
<td>14</td>
<td>\texttt{load_buffer \parallel transmit_word}</td>
</tr>
<tr>
<td>23</td>
<td>\texttt{load_register \parallel generate_parity}</td>
</tr>
</tbody>
</table>

etc.
Parallelism

In this case the time needed to transmit a single word is equal to
\[ \text{load\_register} \parallel \text{generate\_parity} \ (2) + \text{load\_buffer} \parallel \text{transmit\_word} \ (9) = 11 \text{ time units}. \]
This means that the pipelined transmitter is (a little) faster than the non-pipelined transmitter. In this example the trade-off between speed and cost was very explicit: we traded a faster (+9%) execution against an extra cost factor, e.g. the buffer register.

Remark:

Of course it is possible to implement the transmitter in a more optimal manner, for instance by using a serial parity generation mechanism instead of the parallel method used in the example. However, here it was our intention to introduce the notion of parallel processes, and not to design an optimal transmitter.

In a second example we now want to give attention to another important problem encountered in parallel algorithms: the use of limited resources.

4.1.2 Example 4: the multiple adder problem

Suppose we have a (decomposed and sequential) complex system. The control hierarchy of this system consists of a considerable number of levels. The decomposition resulted at the information set level in a great number of primitive operations, some of which are equivalent to each other. As a matter of fact, it happens to be the case that the add-operation (+) is implemented n-times (n\(\geq\)1). This forms, we feel, a considerable waste of circuit elements, and we try to implement the system using a single adder at the information set level. The operands are stored in two dedicated registers, A and B, and the result of the add-operation is stored in a register C. A controller wanting to perform an add-operation then has to execute the following sequence of actions:

\[
\begin{align*}
\text{load\_register\_A;} \\
\text{load\_register\_B;} \\
\text{ADD;} \\
\text{read\_register\_C;}
\end{align*}
\]

This sequence of actions can only be initiated by one controller at a time: it forms a so called critical section or critical sequence.

If the system is purely sequential, which means that only one controller at a time can be active, no problem arises. This is different when the system is, to some extend, parallel. Then it is possible that multiple controllers want to initiate the critical sequence at the same time. Of course, this would result in an incorrect operation of the adder and therefore some means of ensuring mutual exclusiveness between the requesting controllers has to be induced.
In the last two paragraphs we encountered some problems associated with parallel algorithms. In the course of time an extensive theory is developed around these problems: the theory of communication and synchronization between parallel processes. In the next paragraph, the most important results [ANDRE] will be presented.

4.2 Communication and synchronization

As we saw in paragraphs 4.1.1 and 4.1.2, the parallel execution of actions or sequences of actions (processes) is controlled by the cooperative and competitive relations which exist between the various processes. In sample decomposition 3, the concurrently executed actions, e.g. (load_register || generate_parity) and (load_buffer || transmit_word) respectively, cooperated to implement the application: the transmitter.

In example 4, where several parallel processes made use of a single adder (limited resource), the competition between the processes for the use of the adder is obvious: what to do if more than one process request the use of the adder at the same time?

Generally speaking, processes (algorithms) that are cooperating together in order to implement a particular implementation will be aware of each other's existence and their cooperative interactions will be defined explicitly. In contrast, any competitive interactions between processes will be indirect because, in this respect, the processes are unaware of each other.

The mentioned typical cooperation and competition problems can be solved (implemented) by synchronization and communication methods. In sample decomposition 3 this was already evident: the concurrent execution of a pair of actions only was possible if the execution of the preceding pair had finished.

As already mentioned, we consider the processes as taking place in discrete steps. At each step there is an event, which can either be local to the process (subcontroller) in which it occurs, or can have some significance for the whole system, in which it is relevant to the general problem of synchronization. We shall consider only events of this second type, and we lay down the following definition:

Synchronization consists of controlling the evolution of processes, and therefore the occurrence of events, as a function of the past history of the system.

A synchronization problem consists in expressing prior constraints on the possible resultant orderings of the system events. Clearly, only a subset of the events of the system is significant from the
Parallelism

point of view of synchronization; we shall call these sync-points and denote the subset by $E$.

In example 4 this set $E$ of sync-points also can be established. However, to obtain a precise definition it is convenient to associate with every event $A$ (or sequence of events) of non-zero duration two fictitious events of zero duration, $\text{start}(A)$ and $\text{finish}(A)$. We then get in example 4:

\begin{verbatim}
start(ADD);
load_register_A;
load_register_B;
ADD;
load_register_C;
finish(ADD);
\end{verbatim}

In this case, the set of sync-points is given exactly by:

$$E = \{ \text{start}(ADD), \text{finish}(ADD) \}$$

The abstract specification of a synchronization problem now consists of defining:

1. the set $E$ of sync-points;
2. the language of legitimate time-traces, i.e. traces which conform to the time constraints imposed by the synchronization.

In order to implement parallel algorithms we need to express the necessary synchronization constraints and implement them by an 'abstract controller' which may be represented in various ways, e.g. by a special process, or again by several cooperating processes. The controller observes all those events which are significant as far as synchronization is concerned and it has the means available to delay any process in order to produce a legitimate time trace; in the implementation, the fictitious zero-duration events of $E$ are translated into actions performed by the controller.

For any sync-point $A$ we must identify the following:

- the arrival of a process at this point, signalled by a request to the controller;
- the permission given by the controller to the process to continue its execution; this corresponds to the actual point of synchronization;
- the various operations that the controller applies to its internal variables, which may precede or follow this granting of permission.

The abstract specification of a synchronization problem defines:
Parallelism

- a set of variables, representing the states of the processes and/or the resources;
- a set of conditions, functions of the state variables, which determine the way in which the processes evolve;
- the changes that are to be made at the state variables in the course of this evolution.

It is assumed that all operations on the variables are coherent, i.e. those that are applied at each synchronization event can be regarded as instantaneous. Consequently, the implementation must specify:

1. the details of the representation chosen for the variables, such as their location and the number of copies of each;
2. the details of the representation chosen for the operations, i.e. of the sequence of actions which perform them;
3. the way in which the operations are to be executed.

Action (3) must ensure that the required coherence is respected: since the execution time for the sequences of actions associated with the sync-point is not in fact negligible, there can be mutual interference between sequences if precautions are not taken to prevent it.

The implementations can be grouped into two classes according to the representation chosen for the variables: centralized, in which all the variables are held in a single memory and there is only one copy of each; and distributed, in which the variables are shared and distributed in a variety of ways.

A means for mutual exclusion between different sequences of actions seeking to access the same variable has to be provided if the method is to be applicable to all synchronization problems, and this exclusion can apply to messages left in a mail-box, to complete processes or arbitrary sequences of statements. This feature can be used to express the indivisibility of the sequences associated with the sync-points, and thus provides a relatively simple way of ensuring the coherence of the operations. It is not always necessary to impose mutual exclusion on all the operations in order to ensure the coherence of the state variables. Processes can be allowed to execute in parallel and at their own rates, provided that they respect the synchronization constraints and the requirements for the elementary read/write operations to be atomic.

Semaphores, and the primitives P and V associated with them, form the basic tools for mutual exclusion. A semaphore $S$ is a data structure consisting of a counter $s$ which takes integer values and a queue $q$ which is initially empty; the initial value of $s$ can be positive or zero. The operations $P$ and $V$ are defined as follows:
Parallelism

\[ P(S): s := s-1; \]
\[ \text{IF } s < 0 \text{ THEN put the process in the queue } q; \]

\[ V(S): s := s+1; \]
\[ \text{IF } s \leq 0 \text{ THEN activate a process held in the queue } q; \]

These primitives are atomic, guaranteeing that the conditional statement and the associated operations are always executed consecutively.

A process executing \( P(S) \) can find itself blocked if \( s \) is negative or zero, when only a \( V(S) \) operation can release it. If \( s \) is negative, its absolute value gives the number of blocked processes; if it is positive, it gives the maximum possible number of simultaneous executions.

The semaphore makes it possible to implement synchronization expressions in a centralized environment. Access to the synchronization variables or to the controller is protected by defining a semaphore for each variable or group of variables and for the controller, and the processes execute the primitives \( P \) and \( V \), respectively, before and after each synchronization procedure. The semaphores themselves can be used to represent the synchronization variables directly: these will then be summarized in counters taking values which are positive or zero, with processes being blocked when values are zero.

We now are going to study the properties of a distributed implementation: a distributed architecture is a system of connected sites in which each site has its own private memory; there is no common memory. We assume:

1. Every site can communicate directly with every other - i.e. the system forms a logical network;
2. Transmission is error-free;
3. For transmission between any pair of sites \( i \) and \( j \), the order in which \( i \) sends messages to \( j \) is the same as the order in which \( j \) receives the messages. in particular, there is no loss of messages;
4. Any breakdown of a site is detected and signalled to all sites which attempts to communicate with it.

Our problem is how to introduce into such an architecture the abstract machinery which describes the synchronization constraints. A natural solution forms the distributed control: distributing the control over the different sites leads to the need to consider \( N \) control processes, one at each site. Each local controller handles the processes located on its site and all cooperate via the communication system to implement the
Parallelism

abstract synchronization constraints. This distributed control organization gives rise to difficulties.

Due to delays associated with the transmission of synchronization variables between the different sites, the time traces recorded at each site are called partial traces and the real trace is called the theoretical trace. This trace must be a statement in a language $L$ which forms the abstract expression for the synchronization, on which local constraints of sufficient rigor have been imposed to ensure that the legality of the set of partial traces implies the legality of the theoretical trace.

Distribution of the abstract synchronization problem means first of all that a representation of the state of the system has to be constructed, using local variables belonging to each controller. There are three main techniques by which this can be done:

- replication of the set $V$ of variables: actual variables are set up at each site, corresponding to the complete set $V$ and having the same meaning as the 'abstract' variables;

- distribution of set $V$: the set is separated into disjoint parts $V_i$, to each of which corresponds a set of actual variables at a single site;

- decomposition of the variables of $V$: each abstract variable is decomposed into a number of variables which are then located at different sites. An abstract variable is then a function of local variables.

A point of fundamental importance here is that the correspondence thus established between the abstract variables and their local representations is valid only so long as there is no activity in the system. In normal working the consequence of the delays introduced by the transmission of items of information is that the local variables often give only an approximate representation of the true state of the system.

Since the abstract variables forming set $V$ are to be represented by local variables, the abstract conditions governing passage through synchronization points and the modification of the state variables must be expressed in terms of these local variables. The conditions actually applied will not, in general, be identical to the abstract conditions because the local variables will be only either an approximate or a partial representation of the abstract variables. One way of handling the situation is to replace, at each site $i$, the abstract condition by a local condition $\text{condition}_i$ involving only local variables and constructed so as to take account of the relation between the abstract and the local variables. Generally, the local conditions are never weaker than the corresponding abstract conditions.
Before we conclude this section, there are two important issues left: First, it is important that in the design of parallel systems the possibility of reaching deadlock and starvation is considered, and prevented. A system is said to be in deadlock if there is at least one process waiting and no activity is taking place. A process is in a state of starvation, e.g. the process is being delayed for an infinite length of time, if it is never activated, although the system is not in deadlock.

In the second place, we have to consider the case when several processes are waiting for access to a critical resource: which process has to be granted access to the resource first? This decision can be made according to possibly very complex priority rules, which either establish equity or relative priorities between the requesting processes or classes of processes.

In order to make a decision the controller must always know the identities of the processes which have requested but not been granted access, or the class to which these processes belong. A representation of those processes is called a queue. In problems where priorities change with time, several queues have to be maintained, through which the processes progress.

4.3 Consequences for Glushkov's model (3)

In the preceding section we saw that parallelism, e.g. the use of parallel processes, generally causes problems of a cooperative and competitive nature. In chapter 2 we defined a process as a sequence of actions. If we assume that a primitive action is a one-action process, this means that with each initiated action in Glushkov's decomposition model a process can be associated. Two or more processes are parallel if the corresponding master controller initiates them at the same time. Hence, with every controller of the control hierarchy, which is capable of initiating more than one action, parallel processes can be associated.

In sample decomposition 3, the processes load_register, load_buffer, transmit_word and generate_parity cooperated together in a partially parallel manner. The control over this cooperation, was executed by the corresponding master controller, i.e. in this case the main control level. This indicates that the main controller (in sample decomposition 3 the main controller is the master controller corresponding to the mentioned processes) initiated the processes in a partially parallel manner, and by doing so respecting the sync-points of the synchronization problem. The main controller has access to the synchronization conditions transmit_word_ready and generate_parity_ready and by using these synchronization conditions it implements the sync-points.

This knowledge can be expanded to all control levels of the system: the cooperative relations which have to be maintained between the
different processes which are to be executed in a more or less parallel way are controlled by the corresponding master controller. This means that at any level of the control hierarchy a (sub)controller, that concurrently initiates several actions, has control over the cooperative relations between the with the actions associated processes. From this, we can conclude that Glushkov's decomposition model in a natural way incorporates the cooperative relations that might exist between processes in the system. The decomposition of the system in general, and the control hierarchy in particular, is not influenced by the cooperative relationships. The only thing that is changed compared to the purely sequential case is the algorithm implemented by the master controller of the parallel processes.

In example 4, we encountered the competitive relations that can exist between parallel processes. We saw that, whenever the use of a limited resource, e.g. a single action, a sequence of actions (process) or even several parallel processes, can possibly be requested by numerous parallel processes at a time, some means of ensuring mutual exclusiveness between the requesting processes must be provided. The basic tools for this are the semaphores and we study their possible use in our decomposition model by the following example:

4.3.1 Example 5: the use of semaphores

We assume for the moment that we are dealing with a rather complex system, which we have decomposed according to the methods described in paragraph 3.1. An analysis of the primitive actions at the information set level indicates that the shift operation is implemented twice and the add operation (+) three times. These actions are initiated by three different processes (figure 4.5): two multiply processes which implement a multiplication according to the Booth-algorithm, and a distinct process y.

![Figure 4.5 Situation at the lowest control level](image-url)
For the time being we concentrate ourselves on the three add operations and we try to implement the system using a single add operation. This attempt is similar to the situation described in example 4. It means that the three possible sources of add operations, i.e. mult1, mult2 and y, have to use the add-operation in a mutual exclusive way, as indicated in figure 4.6. Otherwise the correct operation of the system would be endangered.

![Figure 4.6 Mult1, mult2 and y use the same add operation](image)

We try to ensure the mutual exclusiveness between the requesting processes by the use of a semaphore `ADD`. As a consequence, the algorithms of the requesting processes have to be enhanced with `P(ADD)` and `V(ADD)` actions, as indicated in the following algorithm:

```
ALGORITHM: mult1/mult2/y;
BEGIN
   
   P(ADD);
   ADD;
   V(ADD);
   
   END;
```

We suppose here that the with the semaphore `ADD` corresponding counter `add` is initialized at one, indicating that only one process can use the ADD-operation at a time.
The \( P(ADD) \) and \( V(ADD) \) operations are defined as follows:

**Algorithm: \( P(ADD) \):**

BEGIN
add:=add-1;
IF \( s<0 \) THEN
push_process; (* put process in queue *)
END;

**Algorithm: \( V(ADD) \):**

BEGIN
add:=add+1;
IF \( s<=0 \) THEN
pop_process; (* activate process in queue *)
END;

For simplicity reasons, it is assumed here that the actions \( add:=add-1, \ add:=add+1, \ push\_process, \ pop\_process \) and the conditions \( s<0 \) and \( s<=0 \) are primitive.

It appears that the actions \( P(ADD) \) and \( V(ADD) \) are complex. Therefore it is necessary to make use of the synchronization conditions \( P(ADD)\_ready \) and \( V(ADD)\_ready \). As an example, the resulting algorithm for \( \text{mult1} \) is given below:

**Algorithm: \( \text{mult1} \):**

BEGIN
... P(ADD);
WHILE NOT(P(ADD)\_ready) DO nothing;
ADD;
V(ADD);
WHILE NOT(V(ADD)\_ready) DO nothing;
...
END;

The corresponding control hierarchy is represented in figure 4.7. In order for the system to operate correctly, the operations \( P(ADD) \) and \( V(ADD) \) have to be atomic, meaning that one and only one \( P(ADD) \) or \( V(ADD) \) operation at a time can be active. For each of the processes \( \text{mult1}, \text{mult2} \) and \( \text{y} \) separately this can be guaranteed: the corresponding master controllers (\( \text{mult1}, \text{mult2}, \text{y} \) respectively)
Parallelism

can accommodate this requirement. In the above-mentioned multi algorithm this has been achieved.

The processes multi, mult2 and y are to some extent parallel and they compete for the use of the add-operation. In this construction it is possible that two or even three of them try to initiate a P(ADD) or V(ADD) operation at the same time. Obviously, some enhancements have to be made to let the system implement the competitive relations, that exist between the parallel processes, correctly.

One way of overcoming this problem is make sure that the P(ADD) and V(ADD) operations are initiated in a single master controller and not, as was the case above, in several different controllers. Each process multi, mult2 and y can send a request for access to that master controller. If the controller is busy serving a request and another request is received, the latter process has to wait until the controller has finished the preceding requests. The serving of a request by the master controller consists of initiating the actions

Figure 4.7 Multi's mutual exclusive control hierarchy
that are part of the critical section and, when the execution of those actions has finished, signalling this to the requesting process by means of a synchronization condition process_served. This system structure is presented visually below:

Figure 4.8 Use of a central server to deal with simultaneous process' requests
Parallelism

From this, the (imprecise) algorithm for the server (master controller) can be constructed:

ALGORITHM: server(1);

BEGIN
  WHILE TRUE DO
    BEGIN
      IF (request_mult1 OR request_mult2 OR request_y) THEN
        BEGIN
          P(ADD);
          ADD;
          V(ADD);
          END
        ELSE
          do_nothing;
        END
    END;

As a consequence, the use of the semaphore ADD brings about the implementation of an extra counter add and queue q.

We can now ask ourselves the question whether or not it is necessary to use semaphores as a means to ensure mutual exclusiveness between requesting processes. The example we just now studied gives rise to the suspicion that we can solve this problem in a more elegant manner. If we examine the example more carefully, we see that the mutual exclusiveness of the processes requesting access to the add-operation is ensured in two ways:

(1) by means of the semaphore ADD and the operations P(ADD) and V(ADD) associated with it;
(2) through the server process. This process registrates the requests from the processes and serves them one after the other. The sequential character of the server process, and the fact that all requests are processed by the same controller ensures the mutual exclusiveness. Despite the fact that the server, or in fact any controller in the control hierarchy, is capable of initiating actions (processes) in parallel, the internal affairs of the server, or, again, any other controller, are processed sequential. This can be exemplified by means of the server controller: if we omit the P(ADD) and V(ADD) operations and use the server controller as the only means of ensuring mutual exclusiveness, the algorithm for the server might look like this:
ALGORITHM: server(2);

BEGIN
WHILE TRUE DO
BEGIN
  IF request_mult1 THEN
    BEGIN
      mult1_served:=FALSE;
      ADD;
      mult1_served:=TRUE;
    END
  ELSE IF request_mult2 THEN
    BEGIN
      mult2_served:=FALSE;
      ADD;
      mult2_served:=TRUE;
    END
  ELSE IF request_y THEN
    BEGIN
      y_served:=FALSE;
      ADD;
      y_served:=TRUE;
    END
  ELSE  
do_nothing;
END
END;

The corresponding control hierarchy is given in figure 4.9.

A problem could arise when a request is entered, while another request is being served or, when several requests are received simultaneously. We can study these situations in the case of the server(2) algorithm given above. Suppose that the server has received a request_mult1 and is busy serving that request. Just after the receipt of request_mult1, i.e. one time step later, another request is received, from mult2. Because the internal operation of the server is sequential, the server proceeds with the serving of request_mult1 and delays the serving of request_mult2. However, when request_mult1 has been served (this is indicated to the requesting process by a synchronization condition served), the server starts the serving of request_mult2. While this request is being served, another request might enter, but the serving of this request has to wait until the current request has been dealt with.

Whenever several requests, for instance request_mult1 and request_mult2, are received simultaneously, the server has to decide the sequence in which the requests are to be served. In the server(2) algorithm request_mult1 is served first, and request_mult2 has to wait until the serving of request_mult1 has finished.
Obviously, the server process is quite capable of handling the requests in an orderly manner, and by doing so establishing mutual exclusiveness between the requesting processes.

![Diagram of control hierarchy of the server(2) algorithm](image)

**Figure 4.9 Control hierarchy of the server(2) algorithm**

In the example another point of interest comes forward: it is possible to introduce relative priorities between the requesting processes. For instance, it the server(2) algorithm request_mult1 has the highest priority (is served first), request_mult2 has a lower priority and request_y has the absolute lowest priority. This priority scheme can easily be altered by changing the algorithm. It is even possible to impose some sort of equity among the requesting processes, for example by cyclically changing the priorities of the requests:
Parallelism

ALGORITHM: server(3);

BEGIN
WHILE TRUE DO
BEGIN
    IF request_mult1 THEN ADD; mult1_served:=TRUE;
    ELSE
        IF request_mult2 THEN ADD; mult2_served:=TRUE;
        ELSE
            IF request_y THEN ADD; y_served:=TRUE;
            ELSE
                do nothing;

    IF request_mult2 THEN ADD; mult2_served:=TRUE;
    ELSE
        IF request_y THEN ADD; y_served:=TRUE;
        ELSE
            IF request_mult1 THEN ADD;
            ELSE
                do nothing;

    IF request_y THEN ADD; y_served:=TRUE;
    ELSE
        IF request_mult1 THEN ADD; mult1_served:=TRUE;
        ELSE
            IF request_mult2 THEN ADD;
            ELSE
                do nothing;

    END
END;

Example 5 can be elaborated even deeper by combining the two multiply processes mult1 and mult2 in the same way as we combined the add processes. In Figure 4.10 this scheme is indicated. Whenever mult1 or mult2 want to perform a multiplication, they send a request to the mult server process. This server process functions globally the same as the add server process: it receives requests and serves them according to some priority/equity rules. Again, mutual exclusiveness is guaranteed by the server process.

The elaboration of example 5 showed us that it is very well possible to incorporate parallel processes into Glushkov's model. The cooperative relations that exist between the parallel processes can easily be maintained by means of an appropriate algorithm for the master controller, which is responsible for the initiation of the parallel processes. We saw that the master controller essentially controls the progress of the processes at the sync-points.
The competitive relations between the parallel processes make it necessary to add special controller processes: the servers. These servers take care of the mutual exclusive relationships in the system. With this knowledge in mind we can extend Glushkov's model:
In the figure only a part of a decomposed system is presented. With respect to the previous model (model 2), only the server controllers are added, at all levels of the control hierarchy (except the main control level and control level 1).

A nice property of the described parallel decomposition model, which we shall refer to as model 3, is that deadlock can never occur. We will now demonstrate this: according to the definition, a system is in deadlock if there is at least one process waiting and no action is taking place. Generally, deadlock occurs in a situation wherein two or more processes are waiting for each other (to do something). In the sequential decomposition model it was clear that deadlock could never occur, because at each moment only a single process was active.

In the parallel decomposition model we introduced just now, several processes can be active at the same time, and therefore we have to investigate the possibility of reaching deadlock more carefully.
Processes can be waiting on the action of other processes in two situations:

1. A controller has send a command to a subcontroller and is waiting for this subcontroller to finish the action;
2. A controller has send a request for access to a server and is waiting for the server to finish the access operation.

In both situations deadlock cannot occur, because

(a) the sequential character of the controllers and servers guarantees that each activated or requested process will be processed (of course this only holds if the concerned algorithms are written correctly);

(b) the hierarchical and top-down decomposition of the system ensures that commands and requests are only send to a controller or server that is part of a lower control level. Therefore it is not possible that a 'cyclic request or command situation' arises, as indicated in figure 4.12. To reach such a situation, which is obviously equivalent to deadlock, it is necessary that commands or requests are send to controllers at the same or a higher control level. This is not possible and deadlock can thus not occur.

As we study the server algorithm carefully, it becomes clear that in the parallel decomposition model (model 3) there is a slight possibility that starvation will occur. For instance, a request made by process y will never be considered if continuous requests are made by the processes mult1 and mult2. Probably this situation will not arise, but potentially it could. An algorithm for the server that guarantees that all requests are served within finite time could well prevent this.

**Remark 1:**

It should be clear by now that the parallel decomposition model is a typical example of a centralized system: the control flow is unidirectional (towards the information set level) and the variables
Parallelism

(state variables in the controllers and information variables in the information set) are unique: only one copy of each variable exists.

Remark 2:

In the case a limited resource is used which has \( n \) entry points, i.e. \( n \) processes can access this resource at the same time, this predicament cannot conveniently be solved by model and the server concept; the use of a semaphore with the initial value of \( n \) is more expedient in this circumstance. However, this situation never arises in the parallel decomposition model, because such a situation will always be resolved by using \( n \) resources with \( n \) entry points in stead of an \( n \) entry point resource.

The knowledge we accumulated in this chapter can be used to adapt the design cycle of digital systems. Roughly, the decomposition stage in digital system design can be divided into two phases:

(1) **The top-down phase.** In this phase the system is decomposed in an purely hierarchical manner (see chapter 3). We descend the system tree, and by doing this we gradually introduce more detailed information into the design. The result of this phase is generally a sequential and far from optimal system: both speed and cost of the resultant system are not acceptable;

(2) **The bottom-up or optimization phase.** In this phase parallelism is introduced, and speed and cost qualities are optimized (to some extend).

The speed of the system can be optimized by executing processes in parallel. Starting at the lowest control level, for every controller (process) an analysis can be made whether or not it is possible to execute the with that particular controller associated (slave) processes in parallel. If this appears to be possible, the algorithm for the corresponding master controller has to be changed in such a way that the sync-points of the originated synchronization problem are implemented.

This procedure has to be performed form the lowest control level up to and including the main control level.

At the same time, when this bottom-up speed optimization procedure is in progress, the cost of the system can also be considered and, possibly, optimized. For that purpose we have to detect the multiple occurrence of processes (and actions) in the system, starting from the information set level. If plural processes are detected, an analysis can be made (on the basis of queuing theory and other statistical methods) whether or not it is desirable to implement the system using a reduced number of these processes. Of course, this means that the remaining
processes have to be utilized in a more or less mutual exclusive manner. If such a decision is taken, a server process has to be included. The server algorithm must specify some sort of priority or equity scheme. This can also be done in a more or less optimal way, according to the results of the analysis.

Again, this cost optimization procedure has to be performed in a bottom-up approach, starting off from the information set level up to and including control level 1.

Naturally, the decisions that we make during these procedures influence each other. For instance, if we decide to implement a process in a mutual exclusive manner, this generally has a degrading effect on the speed of the system. Therefore, we have to consider both speed and cost aspects while making optimization decisions.

A last task that has to be performed in phase 2 is the in chapter 3 discussed replacement of superfluous controllers and synchronization conditions by simple signal lines.

In the next chapter the decomposition of complex systems comes up for discussion.
5. Decomposition of complex systems

The in the preceding sections derived decomposition method/model was, in a natural way, central organized: the control was centralized. As a consequence, this method is not applicable to all digital systems, for as we know there are systems which are intrinsically organized in a distributed way. More precisely, we can identify in a general complex system a number of system parts, which perform a certain function, and by doing so, they act substantially independent from the rest of the system. These autonomous system parts are called functional blocks. Each functional block is an entity, a subsystem. The existence of a number of relatively independent blocks is a direct reflection of the intrinsic distributed character of the system. Indeed, each functional block has its own control part, which is relatively independent of the rest of the system (the other functional blocks).

The decomposition of a complex system into subsystems can be studied by means of a simple example: the decomposition of an imaginary computer system.

Example 5.1:
In this example, we want to decompose a computer system into a number of subsystems. The specification of the computer system immediately suggests the following decomposition into functional blocks: a CPU, a DMA-controller, a UART (Universal Asynchronous Receiver and Transmitter) and a disk-controller. These subsystems perform their respective tasks in a rather autonomous manner. However, a closer investigation shows us that it is again possible to identify within a subsystem a number of functional blocks. For instance, the CPU can be decomposed in an execution unit and a businterface, and the UART can be split up in a transmitter and a receiver.

This example indicates the decomposition scheme given in figure 5.1. There, the decomposition of a general and complex system into a network of autonomous subsystems is presented as a procedure consisting of a number of decomposition steps. At each step (level) the available (sub)systems are functionally decomposed, and the communication between the resulting subsystems is determined on basis of the higher level specifications. This procedure is applied until primitive subsystems are derived: these subsystems are relatively independent of each other, but within these subsystems it is not possible to distinguish several independent functional blocks.

The intrinsically distributed control of the system has been implemented in the network of primitive subsystems. These primitive subsystems are relatively independent, and therefore it is obvious that the internal organization of the primitive subsystems is centralized. It is at this point that we can apply the results obtained in the preceding chapters. The centralized
Decomposition of complex systems

Figure 5.1 The decomposition of complex systems
Decomposition of complex systems

character of the primitive subsystems gives rise to the suspicion that we can use Glushkov's extended decomposition model for a further decomposition of the system. This means that we can use the methods derived in the preceding chapters to make the decomposition complete.

Obviously, the boundary between the decomposition of a system into primitive subsystems and the further decomposition of these primitive subsystems is equal to the boundary between distributed control and centralized control, or the boundary between independent and dependent system parts.

On basis of this knowledge we can adjust the design cycle of digital systems. The decomposition stage in digital system design can be characterized by the following sequence of steps:

1. **Top-down decomposition into primitive subsystems.** This stage is described in more detail above. As a result, the system is split up in a number of primitive subsystems. The control over the competitive and cooperative relations between the primitive subsystems is in a natural way distributed over the subsystems;

2. **Speed/cost optimization at the primitive subsystem level.** The speed of the system can be optimized by the plural implementation of primitive subsystems, so that the execution of particular time critical functions can be accelerated. Conversely, the cost of the system can be optimized by the joint, and thus mutual exclusive, use of primitive subsystems. As a consequence of these optimization transformations, the communication between and the algorithms of the involved primitive subsystems have to be reconsidered.

The decisions we make in this optimization phase have a considerable influence on the finally resulting digital system. Therefore, before we make these decisions, we have to make an analysis and study the possible effects of these decisions. In this way, it is possible to guarantee a fairly optimal system at this stage of the decomposition;

3. **Top-down decomposition of each primitive subsystem according to Glushkov's extended model.** The result of this phase are generally sequential and far from optimal subsystems (see paragraph 4.3);

4. **Bottom-up speed/cost (local) optimization.** Parallelism is introduced and speed and cost of each primitive subsystem are optimized. The cooperative relations between the parallel processes are implemented in the corresponding master controllers. The competitive relations between the parallel processes are implemented in special server controllers. Superfluous controllers and synchronization conditions are replaced by simple signal lines. As a result of this phase, the primitive subsystems are optimal (to some extend) implemented. However, the total
Decomposition of complex systems

system (the network of primitive subsystems) is generally not optimally implemented. This is taken care of in the last step: the global optimization phase;

(5) **Global cost optimization.** If we make an analysis of all the processes (controllers) in the system, i.e. the network of the primitive subsystems, it can occur that a certain process is implemented several times. A thorough investigation of this situation can indicate that it is desirable to implement that controller only once, or at least a smaller number of times. This means that this process has to be shared between the requesting master processes in a more or less mutual exclusive manner. In paragraph 4.3.1 we saw that this property can be conveniently implemented by means of a server controller. In this way, the cost of the system can be optimized globally, i.e. over all primitive subsystems.

An other, slightly different possibility for cost optimization is the following situation: in primitive subsystem A a certain process \( P_a \) is implemented and in another primitive subsystem B a process \( P_b \) is implemented. The processes \( P_a \) and \( P_b \) are very much alike and, although \( P_a \) and \( P_b \) are not exactly the same, it is possible to implement these processes by using a single server controller. This means that the server has to be more extensive and must be capable of controlling both processes \( P_a \) and \( P_b \). Because \( P_a \) and \( P_b \) are similar processes, it is likely that a certain amount of the corresponding slave processes are equal to each other, and therefore it might be worthwhile to share these slave processes by means of the server controller.

As a practical example we can think of a multiplication server which serves requests for signed and unsigned multiplications (figure 5.2).

This global cost optimization phase also has to be performed in a bottom-up fashion. We start with the lowest control level and proceed up to and including the main control level. The result of this bottom-up global cost optimization phase is schematically presented in figure 5.3.
Decomposition of complex systems

Figure 5.2 A multi-purpose multiplication server

Figure 5.3 The global server concept
6. Final contemplations

In this chapter we will accommodate Glushkov's model once again and present on basis of that model a general architecture for complex digital systems.

In paragraph 3.4, in the example of the CPU-controlled transmitter, we derived the in figure 6.1 given configuration. In the example, the CPU acted as a high level controller which was capable of sending a write command to the control automaton of the transmitter. The with the write command associated condition transmitter_ready was, conversely, communicated from the control automaton to the CPU. In figure 6.1 this principle has been applied in a more general sense. The control automaton is capable of communication with the outer world by means of commands and associated conditions. In the original concept only a single com-
mand line was possible, but the server concept makes it possible to implement a number of command lines. To indicate this more general notion, we shall henceforward make use of the word 'request' instead of 'command'.

The operational automaton is capable of communicating information with the outer world. This is done by means of the information inputs and outputs.

![Diagram of Glushkov's model](image)

**Figure 6.2 The final version of Glushkov's model**

If we are dealing with a very simple system, in which only a single functional block (=primitive subsystem) can be distinguished, the in figure 6.1 presented model can be used to depict the system. However, when we are dealing with a more complex system which is constructed of a network of primitive subsystems, we cannot longer make efficient use of this model: the communication between the primitive subsystems cannot be modelled properly. To overcome this problem we present the in figure 6.2 depicted model, in which the insights gained in chapter 5 are used. Globally, there are two types
of communication: internal, between the primitive subsystems, and external, i.e. between the primitive subsystems and the outer world. This division suggests that the in figure 6.1 presented external signals also are present as internal signals. This means that the model of figure 6.1 can be extended with internal requests and conditions between the control automata, and internal information signals between the operational automata. Moreover, we also saw in chapter 5 (see figure 5.3) that requests and associated conditions are communicated directly between two operational automata or between an operational automaton and a control automaton. In the model this is represented by the internal request and condition busses which are connected to the operational automaton.

With this model in mind we can compose the in figure 6.3 given architecture of a general digital system.

The internal communication between the primitive subsystems is represented in the architecture by two sets of busses: the information busses represent the information flow between the primitive subsystems and the request and condition busses represent the control flow between the primitive subsystems. We have chosen for a set of busses to indicate that there is often more than one solution to the decomposition problem. The number of used busses depends on the speed/cost optimization decisions which are made during the decomposition.
7. Conclusion

In this report we presented a structured decomposition method annex model based upon some very important theoretical notions. The development of the decomposition method/model evolved in a natural way, incorporating practical demands and observation results.

The decomposition method we introduced narrows the field of arbitrary design (decomposition) decisions down to an acceptable level. However, for that purpose it is necessary that good analysis tools are developed and implemented.

The step-by-step nature of the decomposition method suggests the possible development of a computer tool, which at least is capable of giving a considerable support to the designer in the decomposition phase of the design. During future research work this will have to be investigated. Furthermore, the manual decomposition of some complex examples will be useful in order to study the properties of the decomposition method more carefully.
Literature


[GLUSHKOV1] V.M. Glushkov Automaton theory and formal microprogram transformation Kibernetica 1 p. 1-9, 1965
[GLUSHKOV2]
V.M. Glushkov
Some problems in the theories of automata and artificial intelligence
Kibernetica 6
p. 17-27, 1970

[GLUSHKOV3]
V.M. Glushkov
Introduction to cybernetics
Academic Press 1966

[HENNIE]
F.C. Hennie
Finite-state models for logical machines
Wiley & Sons 1968

[SEDGEWICK]
R. Sedgewick
Algorithms
Addison-Wesley 1983

[WOOD]
D. Wood
Theory of computation
Harper & Row 1987
ISBN 0-06-047208-1