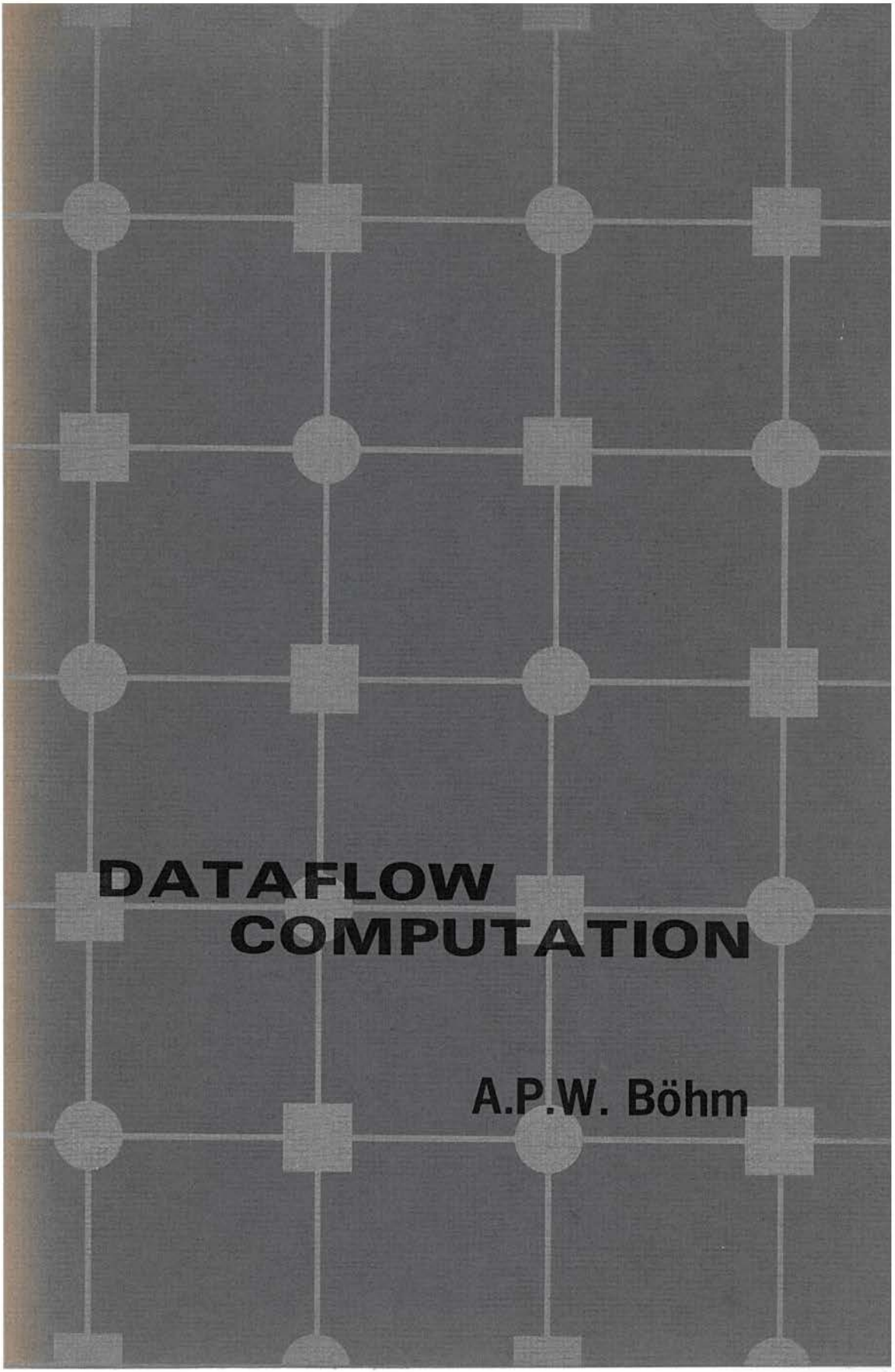

DATAFLOW
COMPUTATION
A.P.W. Böhm

STELLINGEN

behorende bij het proefschrift "Dataflow Computation".

1. In dit proefschrift wordt de "fair merge" operatie geimplementeerd met een dataflow net dat "THERE" boxes bevat. Omgekeerd kan de THERE operatie geimplementeerd worden met een dataflow net dat fair merge boxes bevat.

   hoofdstuk twee van dit proefschrift

2. De "universality" stelling gaat ook op als het waardenbereik van de tokens tot een eindig domein beperkt is.

   hoofdstuk twee van dit proefschrift

3. Zij N een welgevormd dataflow net, S een "start shot" van N, B de verzameling "snapshots" verkrijgbaar door herhaalde "firing" beginnend in S. Het herschrijfsysteem (B, "firing") heeft de "Church Rosser" eigenschap.

   hoofdstuk twee van dit proefschrift

   ROSEN, B.K., *Tree-manipulation systems and Church-Rosser Theorems*, JACM, 20,1 (1973), pp. 160-187.

4. DNP programma's hebben de "encapsulation" eigenschap.

   hoofdstuk drie van dit proefschrift

   FAUSTINI, A.A., *The equivalence of an operational and a denotational semantics for pure dataflow*, Ph.D. Thesis, Report 41, Department of Computer Science, University of Warwick, Coventry, 1982.

5. De formalisering van het begrip "complexiteit van een algoritme" dekt de intuitieve notie die men daarvan heeft niet.

6. Als x procent van de executietijd van een programma verbruikt wordt door vectoriseerbare code, zal vectorisatie hoogstens tot een versnelling met een factor 100/(100-x) van de oorspronkelijke ongevectoriseerde code op dezelfde machine leiden.
In de praktijk zal x met heel veel moeite tot 90 op te voeren zijn.

7. Bij gebruik van gelaagde programmatuur moet uit een foutmelding blijken welke laag die foutmelding produceert.

8. De in de academische wereld opgedane kennis en ervaring met programmatuur wordt te weinig doorgespeeld naar de rest van de samenleving.

9. Programmeeromgevingen kunnen helpen om beter programma's te schrijven maar niet om betere programma's te schrijven.

10. Met het fervent aanhangen of afwijzen van een bepaald programmeersysteem (zoals een programmeertaal) wordt de wetenschap niet verder geholpen.

11. De explosieve groei van de universitaire informatica vormt een bedreiging voor haar kwaliteit.

1 maart 1984 A.P.W. Böhm

DATAFLOW COMPUTATION

dataflow berekeningen

(met een samenvatting in het Nederlands)

DATAFLOW COMPUTATION

dataflow berekeningen
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN AAN DE RIJKS-UNIVERSITEIT TE UTRECHT, OP GEZAG VAN DE RECTOR MAGNIFICUS PROF. DR. O.J. DE JONG, VOLGENS BESLUIT VAN HET COLLEGE VAN DECANEN IN HET OPENBAAR TE VERDEDIGEN OP DONDERDAG 1 MAART 1984 DES NAMIDDAGS TE 2.30 UUR

DOOR

ANTON PEDRO WILLEM BÖHM

geboren op 4 juli 1948
te Rotterdam

1984
MATHEMATISCH CENTRUM, AMSTERDAM

PROMOTOR: PROF. DR. J. VAN LEEUWEN

ACKNOWLEDGEMENTS

A great number of people have helped me during the past few years when I did my research and wrote up this thesis. First of all, I want to thank my promotor JAN VAN LEEUWEN for encouraging me to do research and for working together with me in the first phase of my studies. Chapter two and four are a result of this co-operation. ARIE DE BRUIN introduced me to the semantics of programming languages and played devil's advocate for some of the theorems from chapter two and four. During a number of, sometimes hilarious, sessions we proved my DNP programs correct and studied the formal semantics of DNP. I sincerely hope Arie will sometime write up all the formal stuff I happily left out. PAUL KLINT helped me getting started with his programming language and and compiler writing system. PIM KARS helped me analyse pipeline sort. STEVEN PEMBERTON improved my English. JOHN GURD refereed the thesis. JAN VAN LEEUWEN, DOAITSE SWIERSTRA, ARTHUR VEEN, and HENK PENNING read draft versions. JOKE NOORDWIJK and CORINE DE GEE typed the manuscript and made the vakgroep Informatica a pleasant place to work. The CENTRUM VOOR WISKUNDE EN INFORMATICA and the people working there have always been very special to me. The members of the "dataflow club" (ARTHUR VEEN, JAN HEERING, MARLEEN SINT and PAUL KLINT) provided a critical test-bed for many ideas. TEUS HAGEN gave me the opportunity to use their computing machinery. I am glad that my thesis has been printed at the Centrum voor Wiskunde en Informatica (and will appear as a CWI tract). TOBIAS BAANDERS designed the cover and DICK ZWARST and his crew (JAN, JAAP, JOS and FRANK) were the printers. I would like to thank all these people.

CONTENTS

CHAPTER THREE THE DESIGN AND IMPLEMENTATION OF A HIGH LEVEL DATAFLOW LANGUAGE: DYNAMIC NETWORKS OF PROCESSES

CHAPTER FOUR THE COMPLEXITY OF DNP PROGRAMS

CHAPTER FIVE THE CORRECTNESS OF DNP PROGRAMS

CHAPTER ONE

PARALLEL COMPUTERS AND DATAFLOW COMPUTING

1.1. INTRODUCTION

In the world of computers and computation there are two phenomena that should be in balance but that are not: the supply of versus the demand for computing power. An impressive choice of computing machines is now available. Their possibilities lead people to tackle problems larger and more complex than they ever dreamed of solving before. But when working on these problems, people find out that they need more computing power than there is available. Examples of such problems occur in the fields of meteorology, image processing, global models, windtunnel simulation and the simulation of computer systems ([43],[64]).

It is a recurring concern of computer manufacturers and researchers to find ways of designing faster machines. The speed-up that we have seen during the first generations of computers has been almost invariably brought about by improvements in the technology used for the traditional hardware components. In the traditional von Neumann architecture [15] there is typically one central processing unit connected to one memory, with code and data traveling between them over one channel. Later computers implement the same basic architecture using faster components.

The time has come that the physical limits of this kind of computers are reached. As a compelling example, Hossfeld [43] shows that in a typical family of machines (IBM/Amdahl) the central processing unit has become ten times faster in the nineteen sixties but only twice as fast in the nineteen seventies (see figure 1.1.1.).

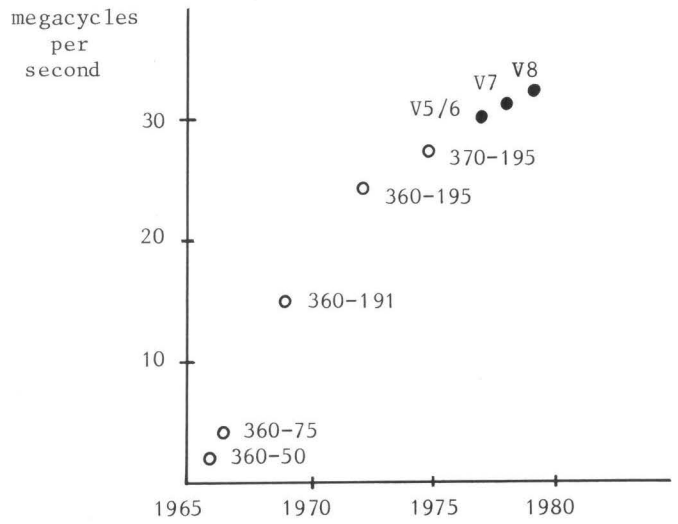


Figure 1.1.1. CPU speed of a typical family of machines

Hockney and Jesshope [40] show that in the period of 1950 to 1975 computer components became a 1000 times faster as measured by gate delay time, whereas whole central processors became a factor of 10^5 faster as measured by multiplication time. The additional speed up was made possible principally by the introduction of parallelism in these basically sequential computers.

Further improvements in computer speed are conceivable only through a radically different approach to computer architecture. This change will lead from basically sequential computer architectures to either parallel (tightly coupled) architectures or distributed (loosely coupled) architectures. We shall focus our attention on the former.

1.2. PARALLELISM

Instead of executing computing tasks one by one : the sequential way, they may often be executed simultaneously : the parallel way. Even when there are more actions involved because of communication and synchronization the

overall compution is likely to go faster, provided that the problem to be solved allows a "parallel" solution at all and that sufficiently many processors are available. Many problems are indeed parallel by nature and computer architects and programmers should be able to make use of this inherent parallelism.

It is not surprising, however, that the sequential way has been preferred for many years: it is easier to understand and (hence) to program, and it has been enforced by the existing hardware. Parallelism, on the other hand, is much harder to understand and may be difficult to capture. The parallelism in a problem may depend on the run-time values of the data, and an additional difficulty is that the amount of communication needed in a parallel algorithm may exceed the amount of calculation in a sequential algorithm. Fortunately the required *parallel mathematics* [87] is now steadily developing and for many problems in e.g. the area of scientific computing the achievable speed-ups through parallel methods are beginning to be understood.

With the advent of highly parallel computer architectures at affordable costs and the maturing insight in the art (and even science) of computer programming, it has become feasible to think parallel in programming.

1.3. PARALLEL COMPUTER ARCHITECTURES

As there is abundant information about parallel computer architectures in the literature ([28],[78],[56],[84]), this overview will be kept short. Underlying each computer architecture there is a model of computation, i.e., a more or less formalized idea of how a computation is to proceed (figure 1.3.1.). For the von Neumann architecture this model consists of iteratively fetching and instruction from memory, decoding it, fetching scalar operands, executing the instruction, and storing a scalar result back to memory. An improvement of this approach is to separate some of these functions and to replicate them in hardware so that they can operate in parallel by looking ahead and executing several instructions simultaneously. The classical example is the design of the CDC6600 [80]. If, like in the 6600, the number of functional units is not too large, the problem of synchronization and interconnection

MODEL OF COMPUTATION	CORRESPONDING ARCHITECTURE
A. Sequential control on scalar data	A1. Von Neumann A2. Multifunction CPU A3. Pipelining
B. Sequential control on vector data	B1. SIMD vector processors B2. SIMD processor arrays
C. Independent, communicating processes	C1. MIMD shared memory multiprocessors C2. MIMD ultracomputers (networks of small machines)
D. Applicative or functional computation	D1. Reduction machines D2. Dataflow machines

Figure 1.3.1. Computer architectures and their underlying computation model.

of these units remains manageable. Also by looking ahead a limited number of instructions, say 3, the possible number of computation orders remains small enough to handle.

A second improved implementation of the sequential control, scalar data model of computation is *pipelining*. Instead of using the same hardware to execute the basic CPU cycle (or any other decomposable task) the cycle is unwound: for every step the appropriate hardware is provided separately ([40], [82]). The gain of this approach depends on the number of steps into which a task can be decomposed.

SIMD (*single instruction, multiple data*) architectures [81] are based on a computation model where the unit of data is a vector or a matrix. SIMD vector processors, such as the CRAYs and the CYBER205, are fast scalar machines extended with special instructions for handling vectors. In SIMD processor arrays, such as the ICL-DAP, there is one control unit but the arith-

metic-logic unit (ALU) is replicated many times. The ALU-s are interconnected in a regular pattern, each has its own local memory and performs the same instruction at the same moment. Such an action may be manipulating local data or communicating with direct neighbours by sending or receiving data.

In a third model of computation there are many independent processes, all operating on their own data. The processes communicate either directly or via shared memory. If the programs in these processors are fixed and simple they can be implemented in VLSI. Systolic arrays [57] are an example of this kind of organisation. In a general purpose machine, complete, independent processors are put together. They communicate with each other by means of a processor-processor or a processor-memory interconnection network. This MIMD (*multiple instruction, multiple data*) approach is by far the most flexible, optimistic but difficult one.

A refinement of the third model is the *applicative* or *functional* model of computation [8]. It compromises *demand driven* and *data driven* computation [84]. In a demand driven computation there is a set of functions which are applied when their results are needed, and a computation starts by demanding the final results. Machines whose architecture is based on this model of computation are called *reduction machines*. A program in such a computer is an expression or function-call demanding the final result. Execution involves evaluating and rewriting this expression. The lazy evaluation concept as known from programming language theory [30] is especially relevant here.

In a data driven computation functions are activated by the availability of their arguments. Since data driven computations are our main interest here, we will elaborate in some detail their underlying data driven model of computation: *dataflow nets*.

1.4. DATAFLOW NETS

Dataflow nets are two-dimensional programs expressing the data dependency between operations. In its most primitive form, a dataflow net is a directed graph in which the nodes represent *processing elements* and the edges represent

data paths. Some data paths will not start at a node (these are the input-lines of the net) or end at a node (the output-lines of the net). Data is presented in *tokens*. Tokens are indivisible, but can be distinguished through an interpretation. They can be transmitted over existing data paths, and processing elements digest them from their incoming edges and send new tokens over their outgoing edges. One *cycle* of a processing element normally consists of the consumption of one token from each incoming edge, followed by the production of one token on each outgoing edge. The execution of a cycle is very similar to a *firing* in the terminology of Petri-nets [69]. The main difference is that processing elements are operators, i.e., token-mappings of some variety.

No assumptions are made about the absolute or relative speeds of the processing elements or about when processing elements take in a new batch of tokens, except that cycles and token transports take finite time. Dataflow computation is completely asynchronous, it implies that tokens may have to queue along a data path if the node at the other end is not processing fast enough or if other inputs of the node are not yet available. However, in some models no queueing is actually permitted and so processing elements will not fire unless all outgoing edges are free.

The many options in specifying a dataflow net have lead to a number of different models. In all models, except in Kahn's [46] and Wadge's ([86], [26]), the processing elements are *token-level functional*. Token-level functionality means that given the same tokens on its incoming edges, an operator will always produce the same tokens on its outgoing edges, independent of the relative times of arrival of incoming tokens and of the state of the computation. Since dataflow computations are asynchronous, no functionality is guaranteed at the global (input/output) level unless proven (see chapter two).

Figure 1.4.1. shows a dataflow net that calculates x^2-4x using primitive boxes DUP (which duplicates any incoming token to both outputs), ↑2 (which produces the square of an incoming value), *4 (which multiplies an input by 4), and - (which subtracts the right input from the left input).

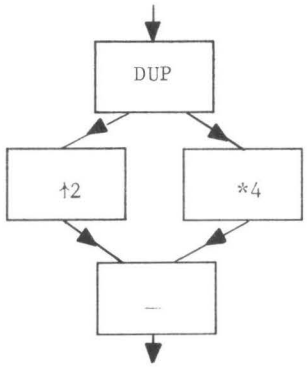


Figure 1.4.1. A dataflow net calculating $x^2 - 4x$.

An execution of the net is pictured in figure 1.4.2., where dots (●) represent the tokens as they are generated and move through the net.

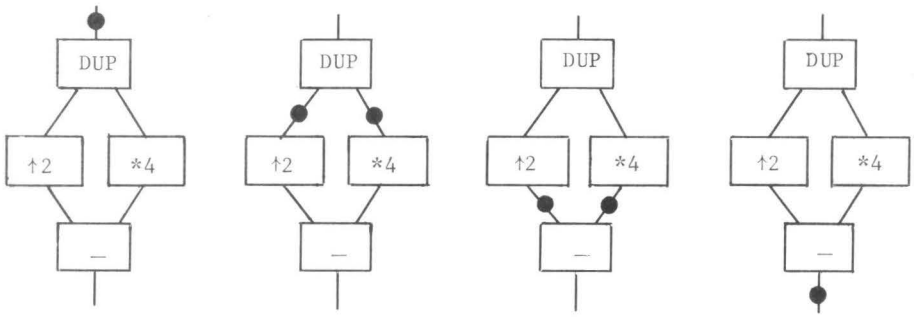


Figure 1.4.2. An execution of a dataflow net.

Karp and Miller [47] have shown that (dataflow) nets with nodes obeying certain rules are deterministic, i.e., the result of executing such a net is independent of the order of the firing of the nodes. The nodes must obey the following rules:

(1) They must consume a fixed number of tokens from their input edges.
(2) They must produce a fixed number of tokens on their output edges.
(3) They must be token-level functional.

These rules are rather severe, though. It is, for example, impossible to have conditional flow of data or loops in these nets. Therefore, all versions of the basic dataflow model that have been developed relax one or more of these rules. If, depending on the value of the input tokens, a subset of the output edges can be selected for firing, it becomes possible to have conditional flow of data. This type of node is called a SPLIT node. In its basic form it has two input edges and two output edges, as in figure 1.4.3.

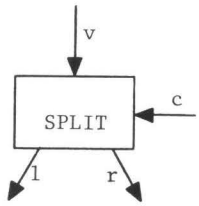


Figure 1.4.3. A SPLIT node.

A token entering via the c-edge has a boolean *control value*. If the c-token is TRUE, the v-token is copied to the l-edge, otherwise the v-token is copied to the r-edge. With a SPLIT node either one of two subnets can be activated, as in figure 1.4.4.

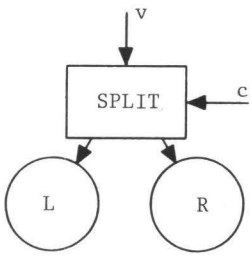


Figure 1.4.4. Conditional activation of subnet L or R.

In order to make the result of the L or R subnet available to a subnet F, a node is needed that selects a subset of its inputs, i.e., that joins the output edges of the L and R subnets. Such a node is called a MERGE (or JOIN) of which there are two types:

(1) A MERGE node with two *data input* edges l and r, and one *control input* edge c. The control value determines whether a token must be consumed from the l-edge or from the r-edge. The l or r token is copied to the output edge (see figure 1.4.5.).

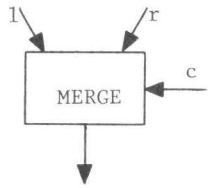


Figure 1.4.5. A deterministic MERGE node.

(2) The second type of MERGE node does not have a control input edge (see figure 1.4.6.),

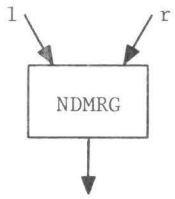


Figure 1.4.6. A non-deterministic MERGE node.

and which input edge the token is to be taken from is decided in some other way. This type of MERGE node is called *non-deterministic* or *time dependent*.

For the moment we will only consider the deterministic MERGE. With SPLIT and MERGE we can now program a conditional assignment such as

z := if c then f(x) else g(x) fi

as shown in figure 1.4.7.

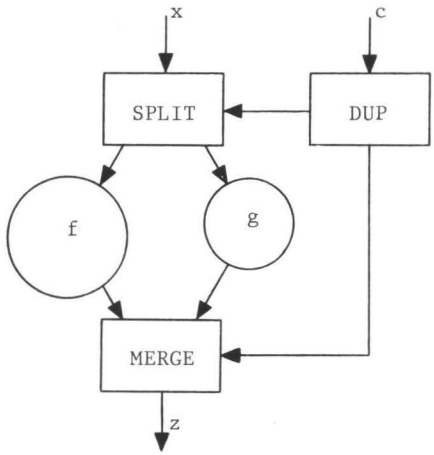


Figure 1.4.7. A conditional assignment.

A loop such as

repeat x := f(x) until g(x)

can be translated into dataflow as shown in figure 1.4.8.

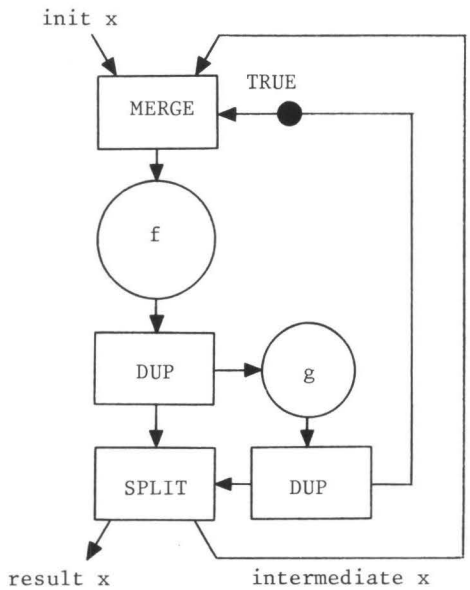


Figure 1.4.8. A loop.

In the net of figure 1.4.8. it appears that we need an initial control value (a "marking") for the MERGE node. We cannot let the first control value come from outside the loop, because then we have to merge the control value from the "outside" and the one from the "inside", which leaves us with the same problem. This phenomenon does not occur when non-deterministic MERGE nodes are used.

Obviously a more complicated computation is translated into dataflow by recursively applying the same techniques. For interest's sake, consider the following program.

```
{
 input(m,n); u:=1;
 while m>0 do if odd(m)
                  then u:= u*n   m:= m-1
                  else n:= n*n; m:= m/2
               fi
            od;
 output(u)
}
```

The dataflow net for the above program is shown in figure 1.4.9., where a SINK node just swallows its input and the POS? and ODD? nodes yield control values. Subnet A controls the loop, subnet B controls the if-statement with subnets C and D implementing the then- and else-part, respectively.

The dataflow net in figure 1.4.9. exemplifies another drawback of the controlled MERGE: even though there will never be more than one token on the two inputs of the MERGE nodes (so non-deterministic MERGE nodes would suffice and would be used in a deterministic way) we have to draw all the control lines and so complicate the net.

In chapter two we will study dataflow nets with non-deterministic MERGE nodes and no control lines. Both their deterministic use (only nets where the two inputs of a MERGE can never contain a token simultaneously) and truly non-deterministic fair merges will be treated.

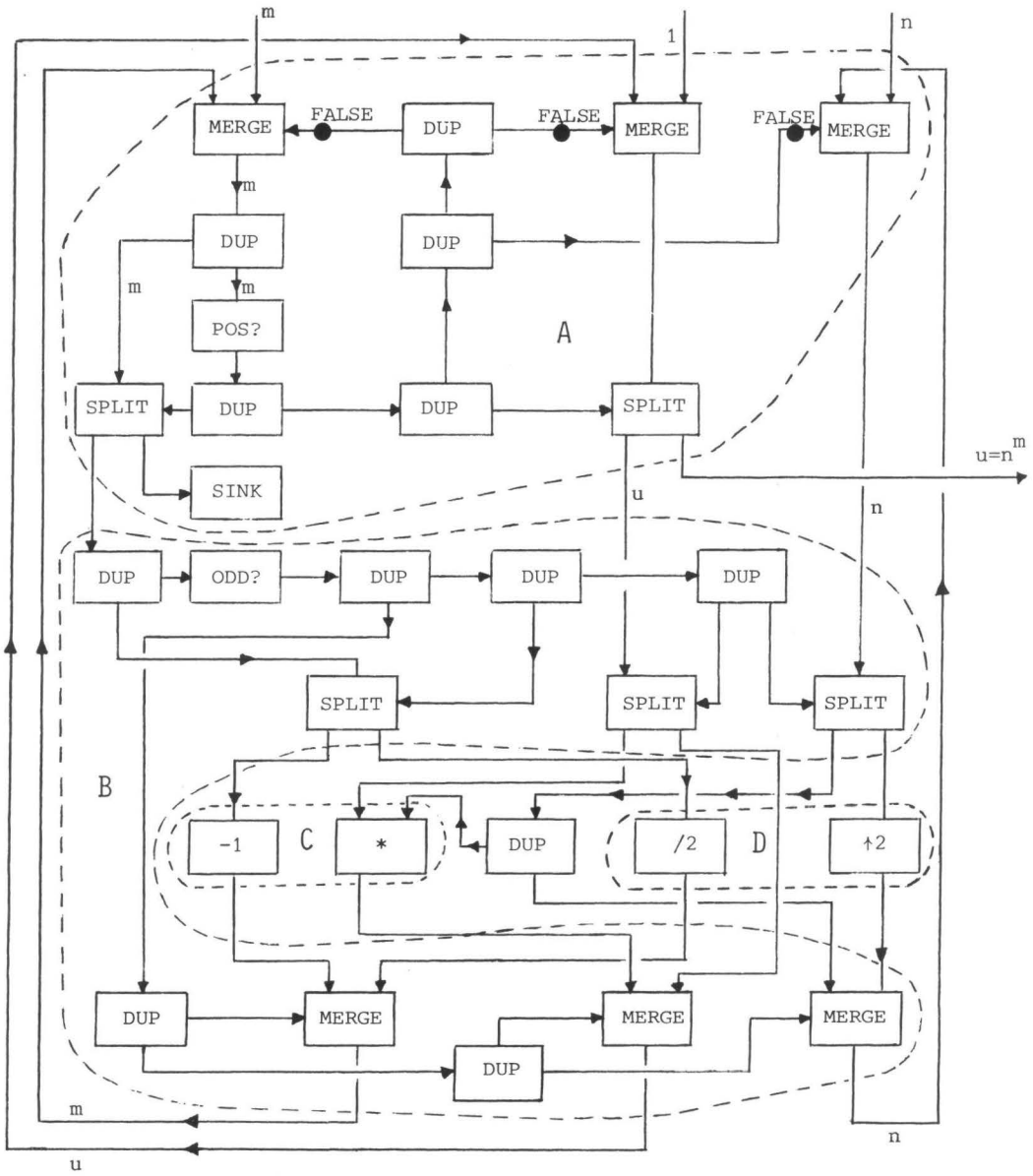


Figure 1.4.9. A complete dataflow net.

1.4.1. Re-entrant use of dataflow subnets.

A subnet inside a loop in a dataflow net may be activated more than once. A subnet can be used to implement a function that will be called at several places in the dataflow net. In both cases, tokens belonging to different computations will flow through the subnet and must not interfere. In chapter two we present a construction that can be used to close a subnet for a new computation as long as the old one is still active. A similar method is used in the dataflow net of figure 1.4.9. Using that construction we can show the computational power of dataflow nets, although a lot of potential parallelism is lost.

If this parallelism is to be saved, simultaneous activations of a dataflow subnet must be allowed while preventing tokens belonging to different calculations from interfering. There are a number of ways to accomplish this. The first requires the edges to behave like queues as we have assumed up to now. This induces an ordering on the tokens, allowing different iterations to be distinguished. This does not guarantee yet that tokens belonging to different iterations do not interfere. The net must be *clean* in that it uses up all tokens it receives. In a second approach the edges are one-token buffers. If, again, the net uses up all its token, a new iteration will push the previous one out of the net. The above methods allow loops to be reactivated in strict sequence. Dataflow models allowing only this sequential cyclic re-entrancy are known as *static dataflow models*.

A more general approach allows both looping and general recursive application of subnets. Again there are two methods. One method permits concurrent re-entrancy via a *call node* which creates a new copy of the subnet every time it is activated. The other method allows the tokens to share the same subnet by ensuring that tokens are passed to the right version of the subnet by some addressing scheme: tokens belonging to different computations are *labeled* or *coloured* differently so that they can be distinguished. Only tokens with the same colour enable a node to fire. In this scheme the edges are just bags of tokens. This method is called *token colouring* or *unraveling*

interpretation of dataflow nets [5]. Dataflow models allowing the general recursive application of subnets are called *dynamic dataflow models*.

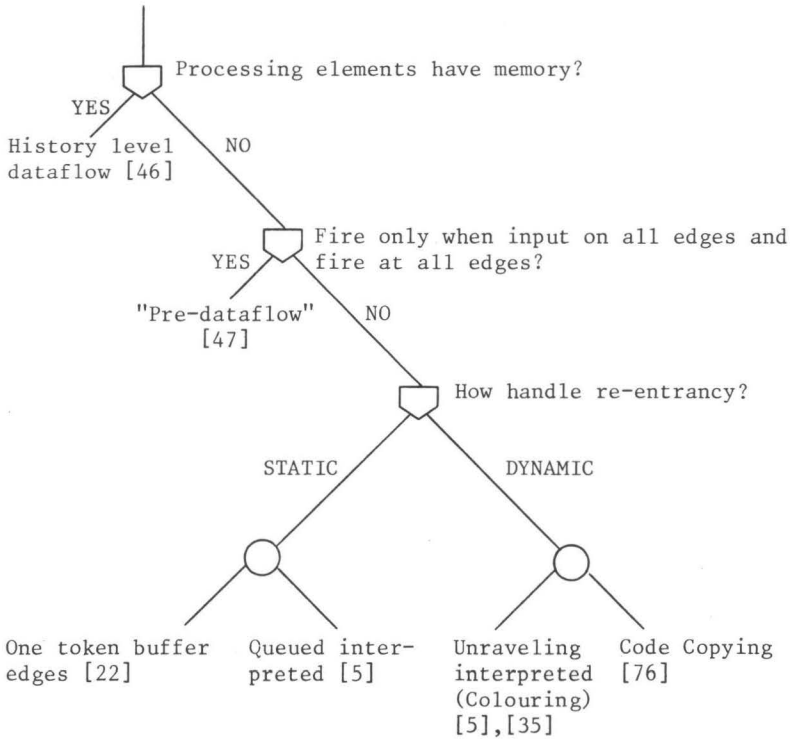


Figure 1.4.10. The various dataflow models.

1.5. DATAFLOW ARCHITECTURES

Having dataflow nets as the underlying model of computation, an unconventional form of computer is required to realize the intrinsic parallelism expressed by it. In [84] an overview is given of the many different dataflow architectures that have been proposed. Experimental programmable dataflow computers are currently under construction at a number of institutions including MIT ([23],[6]), the university of UTAH [19], the university

of Manchester [36] and CER Toulouse [70].

However, there is nothing against implementing a dataflow program by letting nodes be actual processors and edges be wires. A dataflow net thus becomes the specification of an asynchronous special purpose design that may well be suited for implementation on a chip by means of current VLSI technology [58].

1.5.1. An Example: The Manchester Dataflow Machine.

As an example of a typical dataflow architecture, we will discuss the Manchester Dataflow Machine [36] because its design is simple and extensible and clearly shows which problems dataflow does not solve yet, and because some of our results in chapter two relate to it.

The Manchester Dataflow Machine consists of a ring of elements each performing a special task, as shown in figure 1.5.1.1.

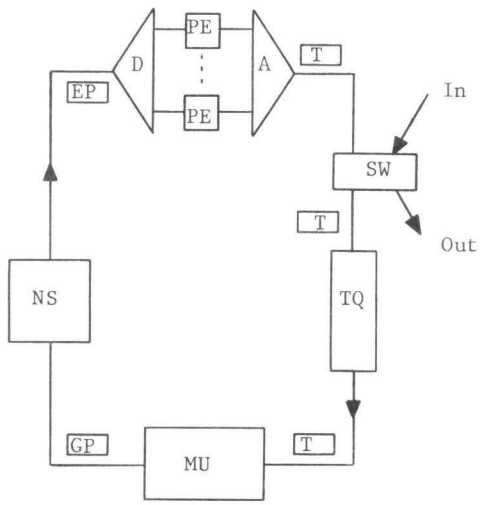


Figure 1.5.1.1. The Manchester Dataflow Machine.

In a general purpose dataflow machine the net representing a particular program cannot be hardwired, and so it must be represented using a data structure of some sort. On the Manchester Machine this data structure consists of labelled nodes containing the function to be performed and the destination node(s) of the result. These nodes are kept in the *node store* NS. In order to execute a node, the node store receives a *group package* GP consisting of a node label and the required operands. The node store then generates an *executable package* EP consisting of operands, the function to be performed and the destination(s) of the result. Executable packages are sent via a *distribution network* D to one of the *processing elements* PE. Processing produces one or more *result tokens* T consisting of datatype, the result value and a destination node label. The tokens are sent via an *arbitration network* A to the *switch* SW.

The switch communicates with the outside world. Result tokens meant for output leave the ring here, input tokens enter the ring and tokens meant for further processing are sent through. The switch sends tokens to the *token queue* TQ, which compensates temporary differences in speed between the *matching unit* MU and the processing elements.

The matching unit is basically an associative memory. Tokens wait here for their partner to arrive, at which time they are put together to form a group package and are sent off to the node store. For efficiency reasons the machine only allows packages containing one or two tokens.

The Manchester Machine actually employs token colouring but for the sake of simplicity we have left the details of this out.

The Manchester Machine makes use of both pipelining (the continuing flow of packages from NS to PE and so on) and low scale MIMD-type parallelism (PE's process different EP's simultaneously). The machine is a truly systolic system: the heart (MU+NS) "pumps" packages to the various "organs" (PE's). The organs use the packages and send the results back to the heart again.

1.5.2. Extensions to the Manchester Dataflow Machine.

1.5.2.1. Global memory.

The virtue of dataflow is at the same time its source of difficulty: there is no global memory. In actual programming, global memory is used in two ways:

(i) It serves as a short term store for intermediate results between instructions or, in dataflow terms, between processing elements.

(ii) It serves as long time storage for information used many times in many places in a program (such as a symbol table).

In case (i) variables can be transformed into data paths. In order to make this transformation straightforward, *single assignment languages* were developed (see section 1.6.1.). However, it has been shown that by building and analyzing their dependence graphs, programs written in a conventional language can be transformed into dataflow nets just as easily ([31],[85], [3],[88]).

Case (ii) is harder because it uses memory in an inherently non-functional manner. In order to mimic global memory, the matching unit of the Manchester Machine is extended so that semi-permanent data can be stored and manipulated there. This is in fact a step back to a von Neumann style memory. The extensions to the matching unit will now be described in some detail.

1.5.2.2. Matching functions.

There are a number of matching functions that can be used to implement time dependent, non-functional, and non-deterministic concepts ([16],[12]). A matching function describes how the matching unit behaves (i) when the partner of a token has already arrived so the match succeeds (the s-action) or (ii) when the partner has not yet arrived so the match fails (the f-action). There are four s-actions and four f-actions.

The operation of the matching unit as sketched in section 1.5.1. was the standard matching function for tokens with a two-input destination. This

matching function is called *extract wait* EW. When the first token for a double input edge node reaches the matching unit it must wait for its partner to arrive, at which point both tokens can be extracted from the memory, combined into a group package, and sent off to the node store. The standard matching function for tokens with a one input destination is by-passing the token store (BY). The full list of s-actions and f-actions now follows.

S-ACTIONS

E for EXTRACT

Both tokens are removed from the token store, packed in a group package and sent off to the node store. This is the standard s-action.

P for PRESERVE.

The token and its partner are packed together and sent off, but the partner remains in the token store. This provides a way to use the matching unit as a memory.

I for INCREMENT and D for DECREMENT.

These s-actions are the same as preserve, except that the remaining token is either incremented or decremented.

F-ACTIONS

W for WAIT.

The token is placed in the token store. This is the standard f-action.

D for DEFER.

The token is not stored. It is sent around the ring "to try again later". This f-action can be used to implement exclusion.

A for ABORT.

The token is not stored. A special token (EMPTY) is sent to the destination to indicate that no partner was found.

G for GENERATE.

Again an EMPTY token is sent to the destination, but the incoming token is stored in the token store on the other input port, so that the next token coming in on the same (original) input port will match it. This f-action can be used to sense the first traversal of an edge.

Apart from BY, the following seven of the sixteen possible combinations of s-actions and f-actions are allowed as matching functions: EW, ED, ID, DD, EA, PG. A token carries a tag indicating which of the matching functions applies.

In section two we will show that there is one basic concept underlying these matching functions: the possibility of checking whether a token has already arrived.

1.5.2.3. A higher level Manchester Dataflow Machine.

The amount of parallelism in the Manchester Machine depends on the number of processing elements. This number cannot be arbitrarily enlarged as the rest of the ring (in particular the matching unit) has a maximum capacity. An extension under consideration [36] is to connect several rings through the switch, which then becomes a full-blown interconnection network (see figure 1.5.2.3.1.). This will make the machine an MIMD machine with dataflow nets as its machine language. Tokens always travel the same distance in this machine, whether they stay in their "own" ring or are transfered to another one. This makes the problem of where to allocate a piece of the dataflow graph much easier.

This design introduces a third level of parallelism, which can be used to implement higher level parallel computation models where the nodes have the computational power of procedures, as in CSP [39], MODULA [89], or Kahn's language [46].

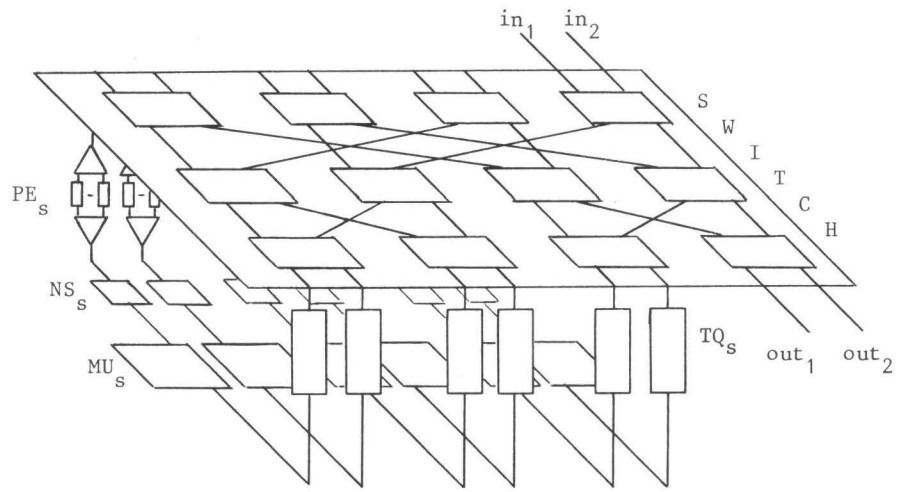


Figure 1.5.2.3.1. A higher level Manchester Machine.

1.6. PROGRAMMING LANGUAGES FOR DATAFLOW MACHINES

1.6.1. Single Assignment Languages.

The languages that emerged together with dataflow machines are based on a *single assignment* principle. There are two versions of the single assignment principle:

(1) An object gets a value assigned to it only in one place in the program.

(2) An object gets a value assigned to it only once during execution of the program [17].

Almost every dataflow research group has its own single assignment language [65]. We will briefly summarize some of the languages. Nearly all of the languages obey the first of the single assignment rules.

LAU [70].
The LAU language (Langage d'Assignation Unique) was designed before the LAU machine was built. LAU incorporates five types of statements: CASE, LOOP, EXPAND (a loop where the programmer can set the maximum number of parallel activations), CALL, and RETURN.

ID [71].
ID is an expression oriented language, supporting abstract data types, streams, and resource managers (a sort of monitors where control resides inside the manager). An ID program creates a large number of parallel tasks called activities.

CAJOLE [37].
A CAJOLE program is a set of definitions. The language supports guarded commands. The programmer can extend the language by defining new syntactic constructs. CAJOLE programs obey the type (2) single assignment rule.

VAL [1].
VAL is an expression oriented language based on CLU [60]. Iteration is viewed as a simple kind of recursion. There are two FORALL constructs. The first generates an array of results, one element per iteration. The second combines the results. There are modules that manipulate streams of data.

LAPSE [34].
LAPSE looks very much like PASCAL, although its semantics is that of a functional language. The language allows array and record structured values and functional subroutines.

LUCID [7].
The motivation for single assignment in LUCID is the ease of program correctness proving. LUCID operators operate on sequences of values.

Single assignment enhances the translation from program text to dataflow net but, as already mentioned in section 1.5.2.1., ordinary "multiple assignment" languages can be translated to dataflow nets as well. The real problem of compiling for parallel machines, which is the allocation of (large) datastructures in parallel memories, has not been solved by the introduction of single assignment languages.

1.6.2. Other languages.

Several research groups are studying the implementation of other languages on dataflow machines such as PASCAL [88], Fortran [45], and functional languages [48].

In chapter three we will study a language for parallel programming. What interests us there is the possibility to express parallelism explicitly at the procedure level and to adapt the parallelism, i.e., the topology of the dataflow net, to the amount and the values of the problem data. The language is called *DNP*, short for Dynamic Networks of Processes.

1.7. SEMANTICS OF DATAFLOW LANGUAGES

The semantics of parallel deterministic languages is based upon the Kahn principle [46]. The meaning of a deterministic net with n edges is described by a set of equations in terms of functions f_i, which specify how the sequence of output tokens on an output edge u_i of some node depends on the sequences of input tokens to that node. The behaviour of the net can be obtained as the minimal fixpoint of these equations. This principle can be extended to non-deterministic models of computation ([67],[26],[11]).

The semantics of token level functional dataflow nets is defined by Arvind and Gostelow [5]. They apply the theory of fixpoint semantics to express the relationship between two different interpretations of Dennis's dataflow nets [22], the queued interpretation and the unraveling interpretation. They show that the unraveling interpretation allows more parallelism than the queued interpretation.

Brock [13] defines the semantics of a dataflow language ADFL, a simplification of VAL. Firstly, a translation from ADFL programs to dataflow nets is defined. Secondly, the semantics of these nets is derived by use of the Kahn principle.

Kahn's semantics and Arvind and Gostelow's semantics differ in the modelling of the traffic of tokens over an edge. The former assumes the edges to behave as queues, while the latter takes token colouring into account. This causes differences in domains and orderings and (hence) a difference in c.p.o. structure.

In both ADFL and Dennis's nets the step from dataflow net to functions is simple because the nodes are token level functional, i.e., they have no inner state. There is a fixed number of node types so their semantic functions can be given beforehand. A similar approach is taken in LUCID ([7],[86]). Here the nodes may have an inner state but as there is a fixed set of node types their semantic functions can still be derived beforehand. This approach cannot be used in a language where the nodes are programmer defined as in the language of chapter three. What is needed then is the definition of an operator from node declaration to semantic function [14].

1.8. DATAFLOW ALGORITHMS

1.8.1. Sequential algorithms.

Computer algorithms can be characterized by the type of program- and data structures they use. When we look at sequential algorithms, the basic program structures are sequence, assignment, condition, loop and procedure call. The basic data structures are scalar, record, array and recursive data structures such as trees and graphs. By analyzing the program- and data structures some of the parallelism from the original algorithm can be reconstructed. As has already been argued, single assignment languages only simplify part of this analysis. Ideally, there is a computer architecture on which the program parallelism, typical for a certain combination of program- and data structures, can be exploited.

Dataflow machines are already suitable for loopfree blocks of conditional assignments, which are hard to run on pipeline or vector machines. The same applies for loops with conditions.

In the present state of dataflow computers it is not yet precisely clear how to implement data structures, such as arrays, while exploiting inherent program parallelism. The combination of matching functions (or their equivalent in other dataflow machines) and higher level architectures seems suitable for tackling this problem. Clearly more research is to be done in this field.

1.8.2. Explicitly parallel algorithms.

No research has been done yet on implementation of programs with explicit parallelism at the procedure level on dataflow architectures. With the advent of higher level dataflow machines this seems to be an interesting research topic. These programs are also interesting for direct implementation in VLSI [58].

In chapter four we will write some explicitly parallel algorithms in DNP, the language introduced in chapter three, and we will also analyse their complexity. The complexity measures will be:

- the number of processes (nodes) in the computation graph,
- the amount of memory in a node,
- the number of edges and the number of tokens on an edge at a certain moment,
- the time needed for the computation.

1.9. SUMMARY OF THE THESIS

In chapter two we explore the theoretical foundation of computation by dataflow. To prove essential properties of dataflow computing we will introduce an elementary model. We prove that for certain, so called *well-formed* nets, asynchronous, parallel execution does not lead to non-functional behaviour, i.e., that all computation orders are equivalent. We prove that our model has universal computing power. The remainder of chapter two is devoted to the simulation of other models of parallel computation.

In chapter three we introduce a high level dataflow language, called *DNP*, based on Kahn's simple language for parallel programming. Parallelism is

explicitly expressible in this language by means of the operation of *expansion*, where a process is replaced by a network of parallel processes.

Chapter four deals with the complexity of some DNP programs and with the expressive power of DNP. We design and analyse algorithms for sorting, matrix multiplication and we will look at the class of divide-and-conquer algorithms. We show that not all computation graphs can be created in DNP. Two ways to overcome this limitation are pointed out. The last part of chapter four is devoted to DNP programs for NP-complete problems.

In chapter five we prove the correctness of some of the programs of chapter four. The proofs are based on the semantics as described by Kahn [46] and formalized by Böhm and de Bruin [14].

CHAPTER TWO

FUNDAMENTAL CONCEPTS IN DATAFLOW COMPUTING

2.1. INTRODUCTION

Models of computation enable us to prove fundamental results about the power and limitations of real or proposed computer architectures. Much of the present theory of computation has resulted from detailed analysis and abstraction of von Neumann architectures. As modern technology is moving away from such architectures we accordingly need to revise our ideas about computation and the way it is performed. In this chapter we shall explore the theoretical foundation of computation by dataflow.

To prove essential properties of dataflow computing, such as the impact of the high degree of parallelism in dataflow nets, we will introduce an elementary model of dataflow computing.

Several dataflow models have been proposed in the past, all based on some notion of a dataflow net. Adams [2] and Rodriguez [73] proposed that four types of primitive nodes be incorporated in the model, namely arithmetical and logical functions, a split node, a controlled merge node and a duplicate node. This set of nodes was adopted by Dennis et.al. [22] and formed the basis of a proposal for a dataflow architecture [23]. Fosseen [29] reportedly proved that these primitives indeed provide universal computing power. Recently Jaffe [44] extended the analysis of Dennis's framework, explored the connections with the theory of program schemata and proved the universality by simulating Turing machine computations in dataflow.

The basic differences between our model and Dennis's model are that our merge primitive has no control input and that we can model time dependent non-functional behaviour by means of a special primitive that reacts to

the (non)availability of a token on one of its input lines. Our primitives are also more elementary. Furthermore, our primitives can be used to model an existing dataflow machine, the Manchester Machine, very naturally.

In section 2.2. we shall define our model. In section 2.3. we shall show that for *well-formed* nets asynchronous parallel execution always leads to functional behaviour, i.e., all computation orderings are equivalent. In section 2.4. we shall define the notion of pipelining and in section 2.5. we shall prove that our simplified model has universal computing power in the sense of computability theory. The proof is very different from Jaffe's and shows direct constructions of dataflow nets for the primitive functions and standard operations from recursive function theory [74]. The main result of section 2.5. will be that for each partial recursive function f there is a dataflow net to compute f that can be used for pipelining, i.e., for producing a continuous stream of result values corresponding to a continuous stream of argument values without the need ever to reinitialize the net. Several applications of this result will be given.

The remainder of this chapter is devoted to the simulation of other models of (parallel) computation with our model of dataflow. In section 2.6. we give a simple simulation of counter machines, which are known to have the same computational power as Turing machines. In section 2.7. we model memory cells. In section 2.8. we model the matching functions of the Manchester Machine. In section 2.9. we model Petri-nets.

2.2. A BASIC MODEL FOR DATAFLOW COMPUTING

A dataflow net is a directed graph in which the nodes represent processing elements and the edges represent data paths. Some data paths will not explicitly start at a node (the input-lines of the net) and some will not explicitly end at a node (the output-lines of the net). Data is presented in *tokens*, which are indivisible, but can be distinguished through some interpretation.

Convention: We shall assume that tokens are natural numbers.

Tokens can be transmitted over data paths only: processing elements *consume* tokens from their incoming edges and *produce* new tokens over their outgoing edges. The combined action of consuming input tokens and producing output tokens is called *firing* or *executing a cycle*. Processing elements are allowed to fire only when all incoming edges have at least one token, with two well-defined exceptions: the JOIN-operator and the THERE-operator (see below). Tokens may queue. If they do, when a processing element starts up a new cycle, it will always pick the front element from each queue on an incoming edge. In systems which do not implement edges as queues, token colouring will be assumed to achieve the same effect.

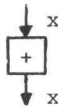
Definition 2.2.1. A dataflow net is said to *compute a (partial) function* $f:\mathbb{N}^k \to \mathbb{N}$ when for all $x_1,\ldots,x_k \in \mathbb{N}$ the following is satisfied: upon receiving tokens representing $x_1,\ldots,x_k$ over distinguished input-lines, the net will eventually produce one token v if and only if $f(x_1,\ldots,x_k)$ is defined, and $f(x_1,\ldots,x_k) = v$.
□

Notice that the net will produce no output if $f(x_1,\ldots,x_k)$ is not defined. The kinds of computation that can be modelled will depend on the primitive operators chosen to build dataflow nets from. We shall use the following primitive processing elements (boxes, operators) as ingredients for dataflow nets:

ZERO: the ZERO-box emits a value (token) 0 once and is then silent forever.

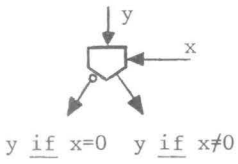
DUP: the DUP-box duplicates any incoming token and emits a copy over both of its outgoing edges.

SINK: The SINK-box swallows and destroys any incoming token.

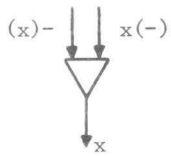


INCR: The INCR-box increments any incoming token by 1, and emits the new value over its output-line.

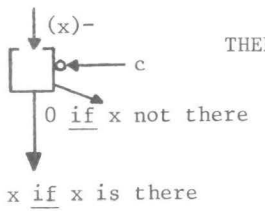
DECR: The DECR-box decrements any incoming token x by 1, provided x>0, and emits the resulting value over its output-line. If x is zero, it is passed on unchanged.



SPLIT: upon receiving the input x and y, the SPLIT-box routes y left or right (i.e., on distinguished outgoing edges) depending on whether x is zero or not (the zero output is encircled).



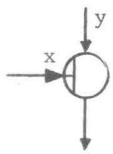
JOIN: the JOIN-box lets any incoming token pass, provided it never finds tokens present on both incoming edges. Otherwise the result is undefined, but we shall always ensure that this does not arise.



THERE: upon receipt of an input c, if an input x is present, it is passed down, otherwise zero is passed to the right.

Clearly the last two boxes may cause problems concerning functionality. The constraint on the use of the JOIN-box removes this problem, because if we allowed two tokens to arrive simultaneously, some decision would have to be taken about which token should pass first. The THERE-box is non-functional by nature and is introduced for that very reason. We will only make use of the THERE-box in non-functional computation models such as memory-cells and the matching functions of the Manchester Machine.

For ease of use we shall introduce one more box, although it is not independent of the primitives above:



GATE: upon receiving tokens x and y, the GATE-box will pass y down.

It is easily verified that the net of figure 2.2.1. implements the GATE-box.

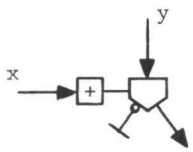


Figure 2.2.1. The GATE.

The rules for building dataflow nets are straightforward. Input lines of the net are connected to input ports of some nodes. Output lines of the net come from output ports of some nodes. With the exception of input and output lines of the net, all input ports are connected to output ports (by "internal" lines). Our notion of (asynchronous) computation by dataflow is identical to that of Adams and Dennis. To exemplify that our nets are primitive but nonetheless powerful, figure 2.2.2. shows a net that implements Adam's controlled merge from section 1.4. Notice that the feedback of the output token ensures the correct use of the lower JOIN-box by preventing a new cycle from starting until the old one has ended.

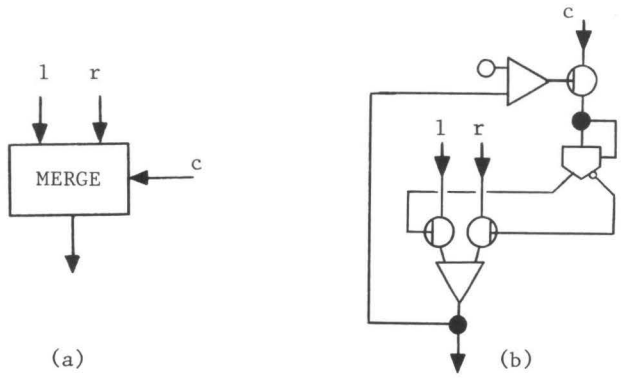


Figure 2.2.2. The controlled merge (a) and its implementation (b).

Definition 2.2.2. A dataflow net is said to be *well-formed* iff:

(i) no JOIN-boxes will ever receive tokens on both their incoming edges simultaneously in any computation by the net, and

(ii) it contains no THERE-boxes.

□

2.3. FUNCTIONALITY

In this section we will consider only well-formed nets. We will need definitions of the following terms: type, in-set, out-set, history, enable, snapshot and execution. Every node in a well-formed net has a *type* $\in$ {ZERO, DUP,SINK,INCR,DECR,SPLIT,JOIN}. The type of a node determines the number of incoming and outgoing edges of the node, and the function it performs. The incoming edges of any node n that is not a JOIN node are called the *in-set* of n. Nodes of type JOIN have two in-sets, the two singletons containing one edge each. The latter convention ensures that the in-sets model the sets of edges that simultaneously enable a node for firing. The outgoing edges of any node that is not a SPLIT node are called the *out-set* of n. For a similar reason to the above, nodes of type SPLIT have two out-sets, since only one of the two outgoing edges will receive a token after firing.

During the activity of a dataflow net, tokens are produced at one end of an edge and consumed at the other. Informally, a *history* is the complete sequence of tokens that have appeared on an edge since a computation started.

Definition 2.3.1. A *history* h is the concatenation of a pair of sequences of values : h = (p of h)^(pc of h). Part p models the sequence of values that have been produced but are not yet consumed, while part pc models the sequence of values that have been both produced and consumed. Parts p and pc are operated upon in queue fashion: producing a new value x causes x to be inserted in p. Consuming a value v causes v to be deleted from p and inserted in pc.
□

The p-part of a history consists of the tokens that are still waiting in the queue associated with the edge. A *snapshot* S (of a dataflow net in action) associates a history S(e) with every edge e.

Definition 2.3.2. An in-set I of node n is said to *enable* n in snapshot S iff for all edges e $\in$ I : p of S(e) is not the empty sequence. A snapshot S enables a node n (S en n) iff there is an in-set I of n that enables n in S.
□

A node of type ZERO is not enabled by any snapshot. We can talk about "the" unique in-set enabling a node n, because we consider only well-formed nets.

Definition 2.3.3. A node n is said to *map* a snapshot S_1 into a snapshot S_2 iff

(i) n is of type ZERO and S_2 is obtained from S_1 by producing a zero on n's output history, or

(ii) n is not of type ZERO and S_1 en n and S_2 is obtained from S_1 by modifying the histories associated to the in-set I of n that enables n and an out-set O of n so that from all input histories of the in-set I of n a value is consumed and on all output histories of the out-set O of n a value is produced according to the function of n.

The resulting snapshot will be written as $S_2 = S_1 n$.

□

Definition 2.3.4. A sequence of snapshots $S_0, S_1, \ldots$ is said to be an *execution* iff

(i) S_0 is a *start shot*, i.e., a snapshot where all histories except those associated with input edges are empty sequences, and where the pc-parts of the histories associated with input edges are empty sequences, and

(ii) for all i=0,1,... there is a node n_{i+1} such that $S_{i+1} = S_i n_{i+1}$.

□

An execution $S_0, S_0 n_1, S_0 n_1 n_2, \ldots$ will be denoted as $S_0 : n_1, n_2 \ldots$ for brevity. For an arbitrary snapshot S and a sequence of nodes φ we say that $S:\varphi$ *exists* if the sequence of nodes can be applied to S in the above sense, without violating the semantic constraints on the JOIN-boxes (i.e., the well-formedness of the net). Note that by e.g. $S:n_1,n_2,n_3$ we denote an execution, while by $Sn_1n_2n_3$ we denote a snapshot.

A moment's reflection at this point shows that dataflow nets in general permit many executions, due to the fact that in a single snapshot many nodes may simultaneously be enabled. Firing nodes in spontaneous order and thus modelling the completely asynchronous behaviour of the net, leads to the question of whether in the end different outputs can result from different (but otherwise permissible) computation orders. In this section we shall prove that this cannot be the case (the "functionality theorem") and that, for all

so called proper executions, well-formed nets display an equivalent behaviour. We need several more concepts before we can give a proof of this.

Definition 2.3.5. An execution E is called *proper*, iff

(i) for every $S_i \in E$ and node n enabled by S_i there is a $j \geq i$ such that $S_j n = S_{j+1}$ (in other words, enabled nodes eventually fire), and
(ii) for every node n of type ZERO there is one and only one S_i such that $S_{i+1} = S_i n$.

□

Notice that after a finite, proper execution the computation in the net is necessarily terminated, i.e., no further node is enabled.

Definition 2.3.6. Given executions E and E', we write $E \subseteq E'$ iff for all edges e and all $S_i \in E$ there is an $S_j \in E'$ such that $S_i(e) = S_j(e)$. (In other words, all histories that occur during E also occur during E'.) E and E' are said to be *equivalent*, iff $E \subseteq E'$ and $E' \subseteq E$.

□

In the following we shall give an argument that all proper executions of a well-formed dataflow net are equivalent. In fact, we shall prove that they can be transformed into one another by "interchanging" actions.

Lemma 2.3.1. *Given a snapshot* S *and two different nodes* n_1 *and* n_2, *then:* $S \underline{en}\ n_1$ & $S \underline{en}\ n_2 \Rightarrow Sn_1n_2 = Sn_2n_1$.

Proof. If n_1 and n_2 are not connected by an edge, the lemma follows immediately, because the sets of incoming edges of one node and outgoing edges of the other are disjoint.

If n_1 and n_2 are neighbours, the firing of one node may concatenate a token to the history associated to an input edge of the other one. Now this firing cannot produce values that are immediately consumed by the other node, because it was already enabled by S, i.e., it had a full set of inputs in an in-set. This is true in particular if the receiving node is of type JOIN: otherwise the well-formedness property of the net would be violated. The tokens that are consumed are therefore the ones that were already there in snapshot S. Consequently $Sn_1n_2 = Sn_2n_1$.

□

Lemma 2.3.2. *Given a snapshot* S, *a node* n *and a sequence of nodes* φ *not containing* n, *then:*

$$S \text{ en } n \;\&\; S{:}\varphi \text{ exists} \Rightarrow Sn\varphi = S\varphi n \text{ (in particular, both exist)}$$

Proof. By induction on $|\varphi|$.

Base: $|\varphi|=1$, the result follows from lemma 2.3.1.

Step: $|\varphi|>1$, write $\varphi = \psi n_1$ (some $n_1 \neq n$)

Because Sφ exists, clearly Sψ exists. And because n does not occur in ψ, the firings of the nodes of ψ can only have caused the p parts of the histories of the input edges of n to have grown without violating the semantic constraint on JOIN nodes. Now observe that Sψ enables both n and n_1 (in case of n by the same in-set as in S). And thus

$$Sn\varphi = Sn\psi n_1 = S\psi n n_1 \text{ (by induction)} =$$
$$S\psi n_1 n \text{ (by lemma 2.3.1.)} = S\varphi n.$$

□

Theorem 2.3.3(The *functionality theorem*). *All proper executions of a well-formed dataflow net that start with the same start shot* S_0, *are equivalent.*

Proof. Let $E = S_0{:}n_1,n_2,n_3,\ldots$ and $E' = S_0{:}m_1,m_2,m_3,\ldots$ be two arbitrary, but proper executions of a given dataflow net. Let $i \geq 1$ be the smallest integer such that $n_i \neq m_i$. Let $S_i = S_0 n_1 \ldots n_{i-1}$ and $S_i' = S_0 m_1 \ldots m_{i-1}$. S_i' enables both n_i and m_i and thus, because E' is proper, there is a smallest k such that $m_{i+k} = n_i$. By lemma 2.3.2. it follows that $S_i' m_i \ldots m_{i+k} = S_i' m_{i+k} m_i \ldots m_{i+k-1} = S_i' n_i m_i \ldots m_{i+k-1}$, and thus that E' is equivalent to the execution $E'' = S_0{:}m_1,\ldots,m_{i-1},n_i,m_i,\ldots,m_{i+k-1},m_{i+k+1},\ldots$ which coincides with E in one more position. Proceeding ad infinitum proves that E and E' must be equivalent.

□

Corollary 2.3.4. *Proper finite executions of a well-formed dataflow net that start with the same start shot have the same length.*

□

In our model, functionality of nets can be interpreted as determinism, when considering the input-history output-history relation of a net. The functionality theorem implies that in well-formed nets we can freely use any proper computation order that is convenient. An execution can be *timed* in different

ways by inserting a *tick* after certain firings. A combination of a certain computation order and a certain timing mirrors the actual running of a dataflow net on some machine. Some interesting computation orders and timings are:

(i) The *sequential timing*. After each firing a tick occurs.

(ii) The *round robin timing*. The nodes are checked in a fixed order. If a node is enabled, it fires and a tick is inserted.

(iii) The *parallel timing*. The execution is rearranged so that if a snapshot S enables nodes $n_1 \ldots n_l$, these nodes will fire first. Now a tick is inserted only after these l firings.

(iv) The *k-bounded parallel timing*. The parallel timing is changed so that if a snapshot S enables more than k nodes, extra ticks are inserted after each k-tuple of firings.

2.4. PIPELINING

Consider a dataflow net as a black box that produces a value $f(x_1,\ldots,x_k)$ a finite time after it has been given its arguments. We want to be able to re-use the net simply by sending it a new set of arguments. We do not necessarily want to wait until a certain computation has finished before sending the new arguments. However, when we look inside the black box, the situation after a computation is likely to be different from the initial situation. This might spoil a later usage of the net. The simplest reason is that a ZERO-box has produced its single token while the next computation also needs one. A second reason is that tokens left behind from a preceeding computation may provide an improper start shot for the next computation. A third reason is that the next set of inputs may interfere with the ongoing computation. In this section we will study the construction of nets that do not have these unwanted properties.

Definition 2.4.1. Consider a dataflow net N computing a (partial) function f. A snapshot S (of N) is said to be *clean* iff any proper execution, starting with S, and extended with a k-tuple $x_1,\ldots,x_k$ of arguments for which f is defined, (on the proper input lines) yields $f(x_1,\ldots,x_k)$. (Observe that the completely empty start shot is clean.) The net N is called *re-usable* if any proper execution starting with a clean snapshot extended with a k-tuple $x_1,\ldots,x_k$ of arguments for which f is defined (i) is finite, and (ii) ends

with a clean snapshot. The net N is said to be *pipelined* if any proper execution starting with a starting shot S_0 consisting of any number of k-tuples of arguments $\underset{\sim}{x}', \underset{\sim}{x}'', \ldots$ for which f is defined, yields a stream of outputs $f(\underset{\sim}{x}'), f(\underset{\sim}{x}''), \ldots$ (in that order).

□

As an example figure 2.4.1. shows four dataflow nets computing f(x)≡0. The net in figure 2.4.1.a is neither re-usable nor pipelined because it will only yield one ZERO. The net in figure 2.4.1.b is not re-usable because any proper execution of the net is infinitely long, but the net is pipelined. The net in figure 2.4.1.c is re-usable but not pipelined because the semantic constraint on the JOIN-box is violated if a next argument comes in too early. The net in figure 2.4.1.d is both re-usable and pipelined.

non re-usable non pipelined	non re-usable pipelined	re-usable non pipelined	re-usable pipelined
(a)	(b)	(c)	(d)

Figure 2.4.1. Computing f(x)≡0.

Consider a dataflow net N computing a function f (figure 2.4.2.):

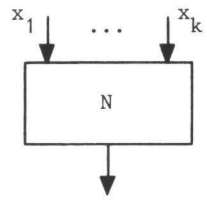


Figure 2.4.2. A net N.

and assume that N is re-usable. Our aim is to make N into a pipelined net, by surrounding N by a "sluice", that will only let a next set of inputs through after the output of the previous computation has been emitted. A sluice network

consists of k upper sluice gates for sluicing in a new k-tuple of inputs and a lower sluice gate for sluicing out a result. Given a re-usable net N the augmentation with a sluice will be denoted as in figure 2.4.3.

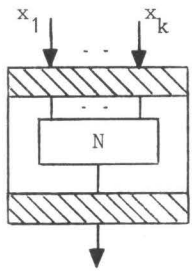


Figure 2.4.3. The sluice construction.

A possible implementation of the sluice is now given. For every input line, the upper sluice gate is as in figure 2.4.4.

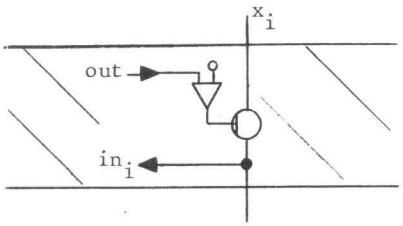


Figure 2.4.4. An upper sluice gate.

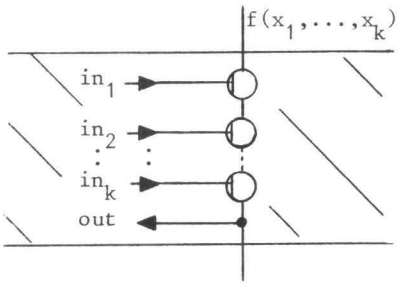


Figure 2.4.5. The lower sluice gate.

The in_i-signal will be sent to the lower sluice gate to report the arrival of a new input token. The output-signal will be sent by the lower sluice gate to report the emission of a result. The lower sluice gate only lets a result $f(x_1,\ldots,x_k)$ through if all x_i-s have been sluiced in. It consists of a series of gates as in figure 2.4.5. The out-signal is duplicated and sent to all upper sluice gates. The idea of letting only entire input-tuples into a (sub)-net was used before by Rumbaugh for the implementation of loops [76], to ensure that one iteration is over before the next one comes in.

Theorem 2.4.1. *Let* N *be a re-usable dataflow network for some function* f. *The augmentation of* N *by the sluice construction yields a pipelined net for* f.

Proof. The construction guarantees that a next set of inputs is not sluiced in until the output from a previous computation is sluiced out. Since N is re-usable this forces a correct use of N, tuple after tuple. The sluice construction also guarantees that, in order for the result to be sluiced out, all the input tokens from the current set of inputs must have been sluiced in. Therefore, no input token can stay behind and interfere with new arguments that it did not belong to.

□

2.5. UNIVERSALITY

We assume that the reader is familiar with Kleene's characterization of the class of partial recursive functions ([50],[20],[62],[74]). An inductive proof that every partial recursive function can be computed by dataflow requires that we prove the stronger result that every such function can be computed by a pipelined dataflow net. For when F, for example, is defined by primitive recursion from g and h:

$$F(0,x_1,\ldots,x_k) = g(x_1,\ldots,x_k)$$
$$F(y+1,x_1,\ldots,x_k) = h(y,x_1,\ldots,x_k,F(y,x_1,\ldots,x_k))$$

then a dataflow computation for F would naturally involve the pipelined use of a dataflow net for h.

Theorem 2.5.1 (The *universality theorem*). *For every partial recursive function* f *there is a re-usable dataflow net* N *that computes* f. *Moreover* N *keeps its queue sizes automatically bounded to* 1.

Proof. By induction on Kleene's formation rules for the partial recursive functions.

(i) the constant-0 function $Z(x) \equiv 0$.

A re-usable net to compute Z was given in figure 2.4.1.d.

(ii) the successor function $S(x) = x+1$.

This function is trivially realized by the INCR-box.

(iii) the projections $\pi_i(x_1,\ldots,x_k) = x_i$ $(1 \leq i \leq k)$.

For any i $(1 \leq i \leq k)$ π_i is realized by a re-usable dataflow net as in figure 2.5.1.

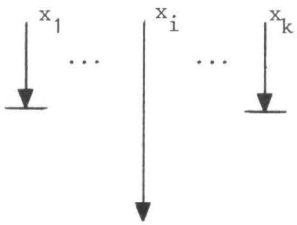


Figure 2.5.1. A net for projection π_i.

The net routes all unused arguments to SINK-boxes to prevent them from interfering with any later computation.

(iv) composition.

Let g be a partial recursive function of m variables and let $h_1,\ldots,h_m$ be partial recursive functions of k variables. Let F be defined by composition from g and $h_1,\ldots,h_m$:

$$F(x_1,\ldots,x_k) = g(h_1(x_1,\ldots,x_k),\ldots,h_m(x_1,\ldots,x_k))$$

Suppose that g and $h_1,\ldots,h_m$ are computed by dataflow nets G and $H_1,\ldots,H_m$ respectively, which satisfy the requirements of theorem 2.5.1. It will be obvious that the net N shown in figure 2.5.2. satisfies the requirements as well and computes F, where the inputs $x_1,\ldots,x_k$ are duplicated and sent to all nets $H_1,\ldots,H_m$.

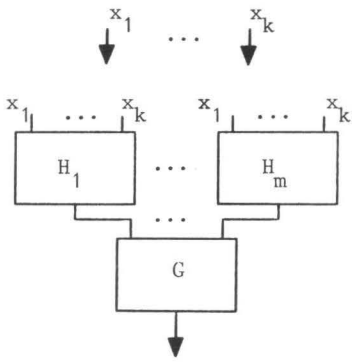


Figure 2.5.2. Composition.

(v) primitive recursion.

Let g be a partial recursive function of k variables and let h be a partial recursive function of k+2 variables. Let F be defined by primitive recursion from g and h:

$$F(0,x_1,\ldots,x_k) = g(x_1,\ldots,x_k)$$
$$F(y+1,x_1,\ldots,x_k) = h(y,x_1,\ldots,x_k,F(y,x_1,\ldots,x_k))$$

Suppose that g and h are computed by dataflow nets G and H, respectively, which satisfy the requirements of theorem 2.5.1. We shall approach the construction of a dataflow net N for F in three stages.

Stage 1: route the input-tokens to G or H, depending on the value of y.

The part of the construction that takes care of this is shown in figure 2.5.3. for the case k=2. (For k=1 or k>2 the construction is adjusted in an obvious manner.) The net for R will be specified later; it is the part of the net where the recursion for y>0 will take place. For y=0 all input-tokens will be gated to G, for y>0 they will all be gated to R. It follows that for y=0 the net N functions as desired, while for y>0 there is no way that the arguments can end up in this same part of the net. Note that the JOIN-box is used properly, since tokens can never come in from both G and R simultaneously, as long as there is no queuing of the inputs. This demonstrates that the sluice construction of section 2.4. to preserve the well-formedness of this dataflow net is needed.

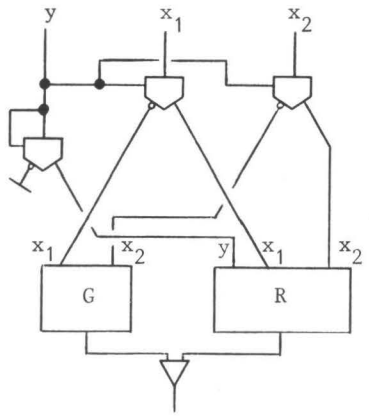


Figure 2.5.3. First design step for N (R remains to be specified).

Stage 2: implement the recursion in subnet R.

R will receive data only when y>0. Its task is to compute and emit the value $F(y,x_1,\ldots,x_k)$. The obvious idea is to compute it by generating the values $F(j,x_1,\ldots,x_k)$ for j from 0 to y, through the pipelined use of H. The main part of the construction is shown in figure 2.5.4. Since H is re-usable but used in a fully pipelined manner, it is surrounded by a sluice. This will guarantee that it sluices in a full set of arguments for every next j. Some care must be exercised so that the various "cycles" (the unspecified subnets in figure 2.5.4.) do not run wild in generating next tuples of arguments for the recursion. In figure 2.5.4. this is arranged by letting H generate a signal whenever another $F(j+1,x_1,\ldots,x_k)$ is produced. The signal is 1 or 0, depending on whether the final j-value (j=y) has been reached or not. The signal is gated to the various cycles. As long as the signal is 0, a next tuple of arguments is generated and gated towards H; this will involve incrementing j by 1 and reproducing every x_i. Whenever the signal becomes 1, the current j-value and the x_i's are gated towards a sink. The signalling guarantees that the recursion is carried out a proper number of times. More importantly, it guarantees that no unneccessary tokens are generated (like j-values larger than y), the queue sizes remain bounded by 1 and that all tokens are removed from the active parts of the net (gated towards a sink) when the recursion is at

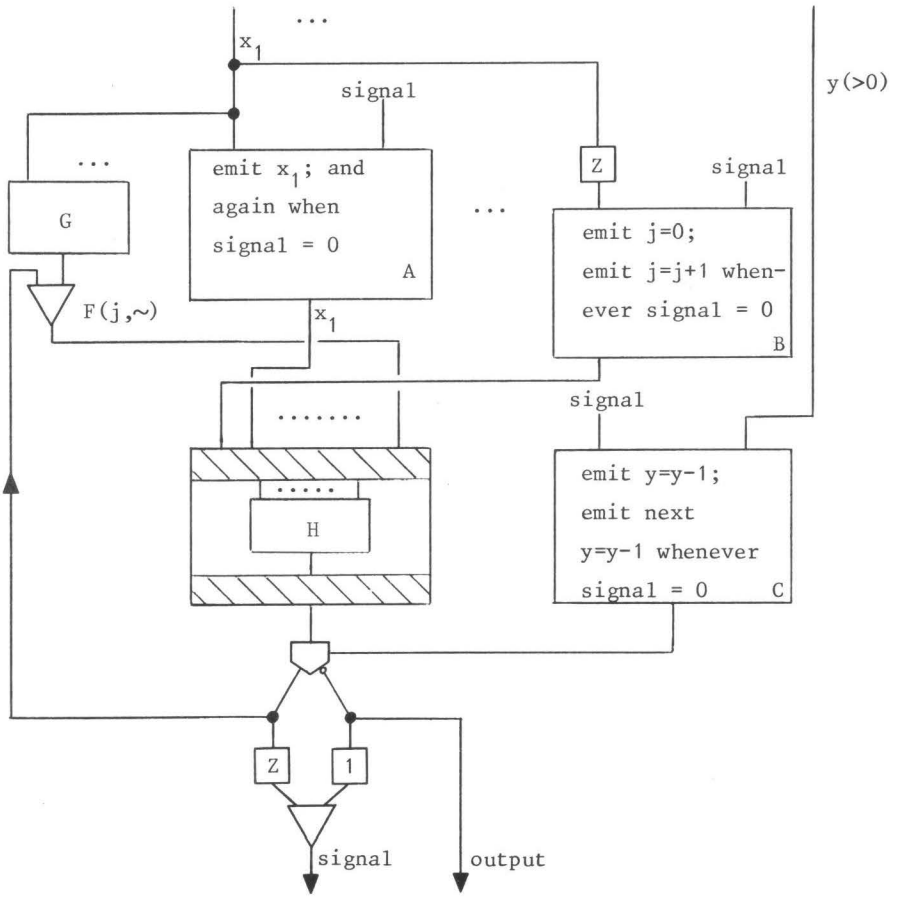


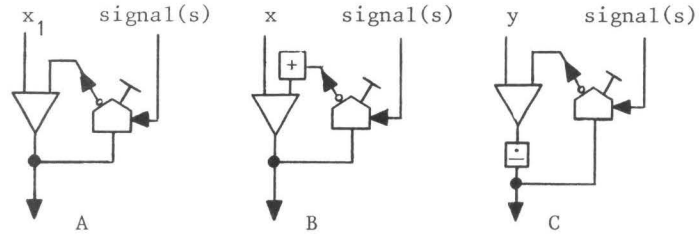
Figure 2.5.4. The R-net.
(Z is the zero function
1 is the one-function)

an end. Provided the remaining parts of the net are correctly specified, R satisfies all requirements for being re-usable! Note that R uses all its arguments since the G and (pipelined) H net do.

Stage 3: fill in the remaining details.

Note in figure 2.5.4. that the JOIN-boxes are correctly used. In particular, there can be no delayed queueing on the incoming edges of the lower JOIN-box, because the signal will be sluiced out by all places that need it

(which, in turn, are sluiced by the H-net which needs a complete set of arguments) every time through the recursion. All we need to do is supply the correct dataflow logic for the unspecified subnets A, B and C in figure 2.5.4. The constructions are all rather straightforward and are shown in figure 2.5.5. Note that nowhere can queue sizes greater than one occur, except at SINK-boxes.



emit x_1 and again when signal=0

emit j=x,emit j=j+1 whenever signal=0

emit y=y-1,emit next y=y-1 whenever signal=0

Figure 2.5.5. Subnets A, B and C of the R-net.

(vi) minimization.

Let g be a function of k+1 variables, and let F be defined by minimization from g:

$$F(x_1,\ldots,x_k) = \mu y\ g(y,x_1,\ldots,x_k) = 0$$

Suppose that g is computed by a dataflow net G that satisfies the requirements of theorem 2.5.1. We shall construct a re-usable dataflow net for F.

To compute F, we shall implement the straightforward idea of computing the values $g(j,x_1,\ldots,x_k)$ for j from 0, until a value 0 is encountered. The construction of a dataflow net for it is shown in figure 2.5.6. Since G is obviously used in a pipelined fashion, it is surrounded by a sluice construction. As long as the g-value remains non-zero, a next j-value will be generated and gated to G, together with a next set of copies of x_1 to x_k. To keep the cycles in the net from running wild, we again use a signal that is tested after each g-value is generated. The signal will be set to 1 or 0, depending on whether the g-value is 0 or not. When the signal is 0, it will trigger the generation of a next set of arguments for G. When the signal is 1, it will direct the current j-value and the cycling x_i-values to sinks and, thus,

reset the A and B boxes. At the same time, the current j-value is sent down the output line of the net as the result of the computation. Notice again that the queue sizes remain 1. The A and B subnets are already specified in figure 2.5.5.

□

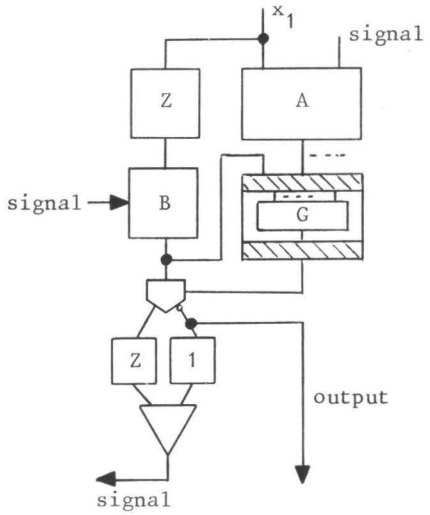


Figure 2.5.6. Dataflow net for minimization.

Together with theorem 2.4.1., theorem 2.5.1. immediately implies the following theorem.

Theorem 2.5.2(The *pipeline theorem*). *For every partial recursive function* f *there is a pipelined dataflow net* N *computing* f *that uses no queues of size greater than one.*

□

It follows that dataflow nets, as defined here, provide yet another basis for computability theory. We note on the other hand that every well-formed dataflow net can be simulated by a deterministic Turing machine. No non-determinism is needed to guess which box will fire at any particular moment, because by theorem 2.3.3. we can choose a fixed computation rule.

From the pipeline theorem one can immediately derive a number of undecidability results for dataflow computing. We shall mention only one.

Theorem 2.5.3. *Well-formedness of dataflow nets is undecidable.*

Proof. Suppose well-formedness were decidable. Consider a dataflow net as shown in figure 2.5.7., where we allow f to be any partial recursive function. A net of this sort is well-formed iff f is everywhere undefined. But the latter is known to be undecidable.
□

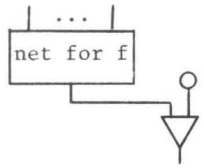


Figure 2.5.7. Well-formedness is undecidable.

A conclusion is that well-formedness, like correctness, can only be ensured through a precise and disciplined construction procedure for dataflow nets. There is a second conclusion to be drawn from 2.5.1. Well-formedness and functionality of a dataflow net are, in a certain sense, equivalent concepts (see section 2.2.). Hence the functionality of a dataflow net is undecidable just as the functionality of a nondeterministic Turing machine is undecidable.

Finally, we shall give an application of the pipeline theorem related to the generation of sets. Hitherto only a few examples were given of dataflow nets which emit sequences of numbers of a specified kind in a specified order [86]. Very generally we can now state the following.

Theorem 2.5.4. *For any recursively enumerable set* S *there is a dataflow net that generates the members of* S *in enumeration order. Moreover, the net does not need any queue sizes to be larger than* 1.

Proof. It is well-known [74] that any non-empty r.e. set S is the range of a total recursive function F. Thus to enumerate S by dataflow, all we need to do is feed the arguments 0,1,2,... into a re-usable dataflow net for F. The construction is shown in figure 2.5.8. The sluice construction is modified here in that it also generates the input values for the net for F.
□

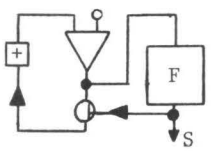


Figure 2.5.8. Generating a non-empty r.e. set S.

2.6. TURING MACHINE SIMULATION

Jaffe [44] has given a direct simulation of a Turing Machine by means of dataflow. We will present here a (more straightforward) simulation of arbitrary counter machines which in their turn can simulate an arbitrary Turing Machine [42].

A counter machine consists of an input tape, a finite control, and a number of counters. A cell on the input tape contains a 0 or a 1. The whole tape contents is enclosed by a begin-of-tape-mark and an end-of-tape-mark. (These marks are represented by numbers unequal to 0 or 1.) A counter can hold a nonnegative number in unary representation: 0,01,011,... A transition of the machine consists of performing either a *read* or a *counter-manipulation*. If a read is performed, the next state in the finite control depends on the current state and the symbol read. A counter can be *incremented*, *decremented* or *tested for zero*. The next state after a test for zero depends on the current state and the result of the test. In any case there are at most two possible next states of a certain state.

Theorem 2.6.1. *For every counter machine there is a well-formed dataflow net simulating it.*

Proof. We will construct a dataflow net for a given counter machine. The net will be built from certain types of subnets. To avoid uninteresting details, we will only give the functional specification of these subnets.

The input tape is available on the only input line of the net. The whole input is read and converted to an integer. This conversion is performed by a special subnet CTI shown in figure 2.6.1. The subnet CTI sends one token tc, representing the tape contents, to a subnet PCM that will simulate the particular counter machine.

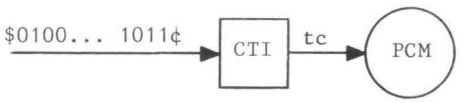


Figure 2.6.1. First design of the counter machine simulation.

The subnet PCM faithfully mimics the finite control and counters of the particular counter machine. For every state in the counter machine there is a subnet which is activated by sending it the (rest of the) tape contents. If a state performs a read it will decode the input token into a symbol (0 or 1) and a next tape contents. (See figure 2.6.2.) Reading from an empty tape will cause no token tc to be produced.

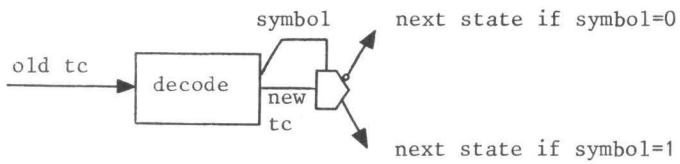


Figure 2.6.2. A read state.

A subnet for a counter-manipulation state sends an *opcode* (say 0 for decrement, 1 for increment, 2 for test for zero) and its *state-number* to the subnet representing the counter. The counter subnet executes the opcode and distributes the result (say 0 for acknowledgement of decrement and increment, and for a zero result of a test for zero, 1 for a non-zero result) back to the counter-manipulation state.

Just as in "real" counter machines the counter value is maintained in a unary representation, i.e., as a sequence of 1-s and one 0. This sequence resides on an edge that is both input and output to the counter subnet. Incrementing is done by producing a 1; decrementing by reading a token. If the token was 0, it is put out again and a next token is read. If that is a 0 again, the counter value was zero. The 0 is put out again so decrementing zero yields zero. Testing for zero is done similarly.

After a counter manipulation, all tokens are sluiced out in order to prevent the counter value from spreading around the various parts of the counter subnet.

□

2.7. MODELLING MEMORY

In this section we will show that dataflow allows the design of general memory cells. It does not follow directly from the universality of dataflow nets that memory cells can be built, because they are inherently non-functional at token-level. We will study the design of two types of memory cells:

(i) the first type of memory cell, called *Memo1*, has two inputs and one output (see figure 2.7.1.). The c-input line carries control values which determine whether a *retrieve* or a *store* is to be performed. If a store is to be performed, the cell will consume a token from the d-input line. If a retrieve must be performed the cell produces the token it has last read in, on its w-output line.

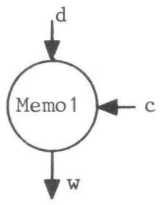


Figure 2.7.1. A history-level functional memory cell.

Clearly, this cell is history-level functional, i.e., upon receiving the same sequences of c- and d-values it produces the same sequence of w-values. We can achieve this by designing a well-formed dataflow net for Memo1.

(ii) the second type of memory cell connected to the outside world by m store input lines, n retrieve input lines, and n write output lines (see figure 2.7.2.). We call this a *Memo2* cell. If the cell receives a token over its i-th retrieve input line it will produce its memory contents on the i-th output line. If the cell receives any store input token it will store the token as its new memory contents. If inputs arrive simultaneously, they will be merged fairly but non-deterministically.

When we connect the i-th store line to a *writer* subnet, and the j-th retrieve and write lines to a *reader* subnet, the similarity with the well-known readers-and-writers problem from operating systems theory [38] becomes obvious.

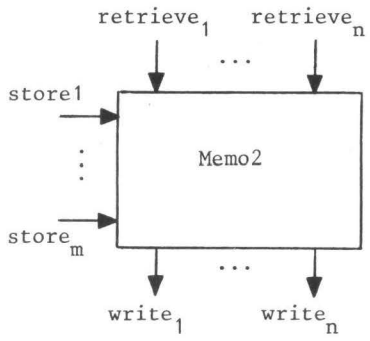


Figure 2.7.2. A nondeterministic memory cell.

The design of the Memo1 cell is straightforward. Its dataflow net is shown in figure 2.7.3., where the CM-subnet is the controlled merge net of figure 2.2.2.

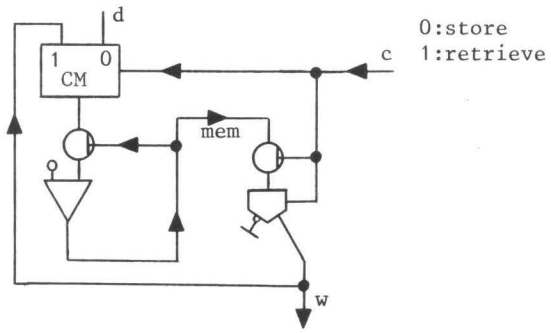


Figure 2.7.3. The Memo1 cell.

The contents of the memory cell is waiting for a c-signal to release it. If a store is to be performed (c-input = 0), the old contents is sent to a SINK-box and a new d-token is let in. If a retrieve must be performed (c-input = 1) the memory value is put on the w-output line and cycled back into the net. Notice that the net is well-formed. If a retrieve is performed before any store, the net will output a zero.

Now if we want to design a dataflow net for a Memo2 cell which allows simultaneous stores and retrieves we can no longer avoid time dependence or history-level non-functionality: because all well-formed dataflow nets are functional (according to the functionality theorem) there cannot be a well-formed net implementing Memo2.

The building block needed for implementing a Memo2 cell is a *non-deterministic fair merge* FM. This is a subnet with two inputs and two outputs (see figure 2.7.4.).

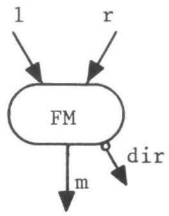


Figure 2.7.4. A non-deterministic fair-merge.

The FM-subnet must operate according to the following specifications:

(i) If a token arrives at either the l-input or the r-input, the token is passed onto the m-output and a token representing its input direction is emitted on the dir-output (r=0, l=1).

(ii) If there are tokens on both l-input and r-input one of them is chosen non-deterministically to be passed onto the m-output and its input direction is again reported on the dir-output. The other input token is preserved.

(iii) If a token arrives, it will be consumed within a finite number of time-steps, where time-steps are measured in terms of firings of basic processing elements.

Part (iii) of the above specification is important and we will name it the *fairness-property*.

Using FM-subnets and a Memo1 cell we can implement a Memo2 cell. Figure 2.7.5. shows a Memo2 cell with one retrieve and one store.

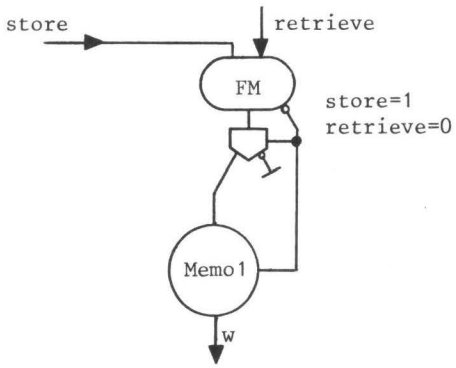


Figure 2.7.5. Memo2 cell with one store and one retrieve.

Memo2 cells with more stores and retrieves are designed similarly, but now there are *fan-in* and *fan-out* trees to direct the inputs to the Memo1 cell and the outputs to the right output lines. Fan-in trees consist of FM-subnets. Fan-out trees consist of SPLIT-boxes. The various dir-outputs of the FM-subnets fanning in the retrieves are used to control the SPLIT-boxes in the fan-out tree. As an example figure 2.7.6. shows a Memo2 cell with four retrieves and two stores.

Clearly, for every FM-subnet in the tree that fans in the retrieves, there is a SPLIT-node in the tree that fans out the various writes. The dir-line of the i-th FM-subnet of the j-th level of the fan-in tree is connected to the control input of the i-th SPLIT-box of the j-th level of the fan-out tree. A moment's reflection may be needed to see that a result token will, on its way out of the net, meet the dir-tokens that were fired when the retrieve token that caused the result token to be written passed a FM-subnet.

The rest of this section will be devoted to the implementation and fairness proof of the FM-subnet. The difference between a FM-subnet and a JOIN-box is that the FM-subnet must sense the arrival of an input token in order to mutually exclude simultaneous arrivals, and it must implement a fair scheduling

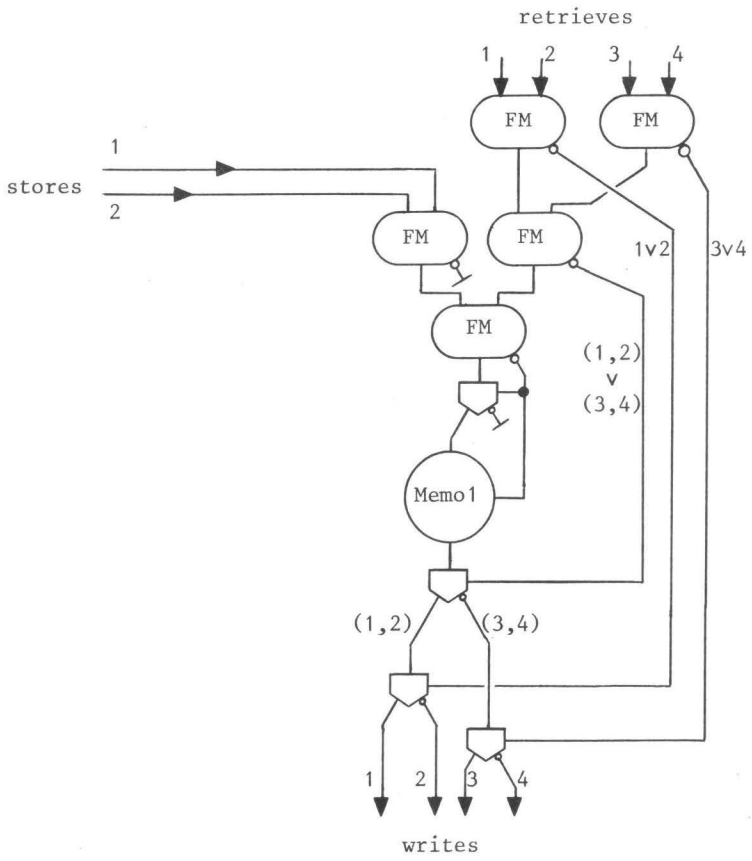


Figure 2.7.6. A Memo2 cell with more stores and retrieves.

algorithm to prevent a token that has arrived from waiting indefinitely long. Sensing the arrival of a token and acting upon arrival and non-arrival can be done using the THERE-box. The FM-subnet is shown in figure 2.7.7.

The thick lines in figure 2.7.7. carry the data from left or right input to m-output. The thin lines carry control-data needed to exclude left and right, implement fairness and generate the dir-output. Notice that at any moment at most one control token exists. The control token, initially generated by the ZERO-box, cycles around between the two THERE-boxes until an input token arrives at the left or right input. The input token is emitted on the m-line and its incoming direction is reported on the dir-line. After

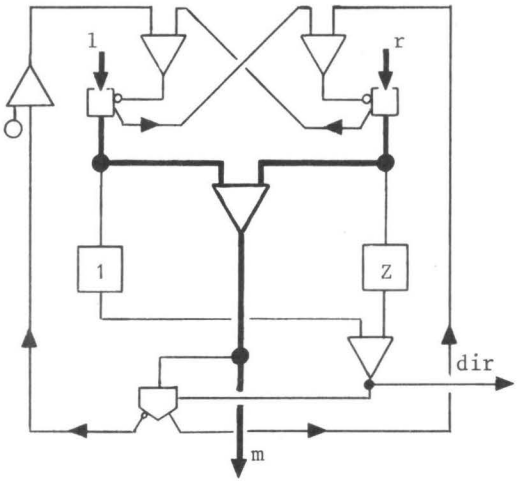


Figure 2.7.7. The FM-subnet.

the dir-token and m-token are dispatched a new control token is generated. If a left input was selected, the right THERE-box will receive the control token first and vice versa.

Theorem 2.7.1. *A token arriving at an input of the* FM-*net will pass through the subnet within a finite number of time-steps (in other words, the* FM-*net is fair).*

Proof. First notice that the JOIN-boxes will never receive tokens at both inputs simultaneously, because at most one control token will exist at a given moment. Let ?L (?R) denote the arrival of a control token at the left (right) THERE-box. Between a ?L (?R) event and a ?R (?L) event there are only a finite number of time-steps, because either there was no input at the left (right) THERE-box and a control token was sent (via an upper JOIN-box) to the right (left) THERE-box, or there was an input and within a finite number of time steps the input token has gone through the net and has generated a control token that was sent to the right (left) THERE-box. A token that arrives at an input will therefore pass through the THERE-box and consequently through the whole FM-subnet within a finite number of time-steps.

□

Corollary 2.7.2. *Using the components defined in section 2.2. one can build memory cells with any number of store and retrieve lines.*

□

2.8. MODELLING THE MANCHESTER MATCHING FUNCTIONS

In this section we will show that the matching functions of the Manchester Dataflow Machine defined in section 1.5.2.2. can be implemented in dataflow directly, although in the actual machine there is a special piece of hardware, the matching unit, that performs these functions. In our model the matching function is performed by a dataflow subnet that is placed in front of the target node, except (of course) the standard matching functions EW and BY (see figure 2.8.1.).

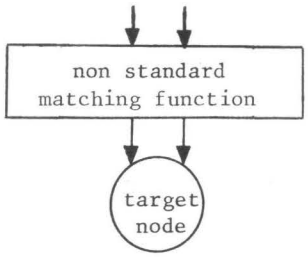


Figure 2.8.1. Implementing a special matching function.

We will only implement the more interesting matching functions ED, PD, EA and PG. The left input carries the special matching function. An EMPTY-token is represented by a 0 over a special output line.

ED: EXTRACT DEFER (success: put out both tokens,
failure: recycle the left input token)

The dataflow net for the ED-matching function is shown in figure 2.8.2. When a left input token arrives there is either a right input token available or not. If the right input is available both tokens are passed (s-action extract), otherwise the left input token is sent back and is merged fairly with other incoming left input tokens (f-action defer). Recall that the THERE-box emits a zero on the no-line if there is no input. The 1-subnet emits a one every time it receives a token.

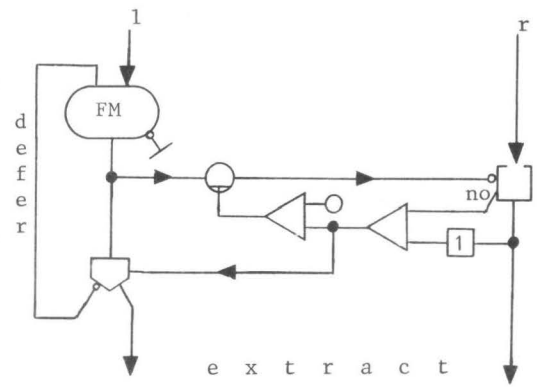


Figure 2.8.2. The ED-matching function.

PD: PRESERVE DEFER (success: put out the left input and the memory token, failure: recycle the left input token)

The dataflow net for the PD-matching function is shown in figure 2.8.3. When a first left input token arrives there is either a right input token or not. If there is no right input token, the left input token is sent back

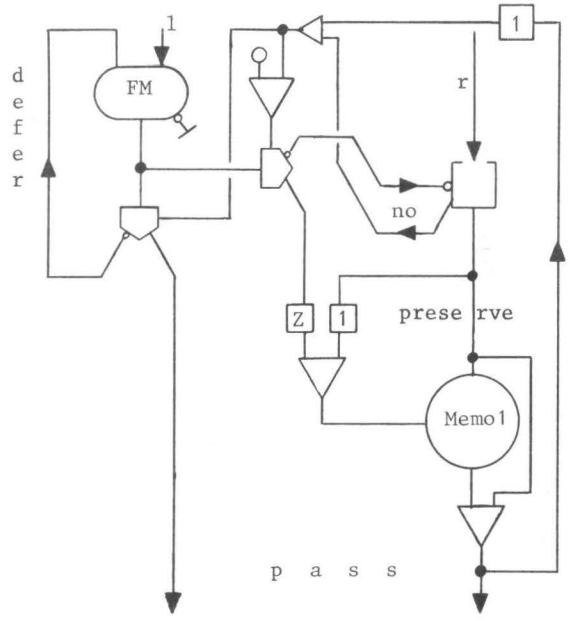


Figure 2.8.3. The PD-matching function.

to the input of the net and fairly merged with other left inputs. Subsequent left input tokens are dealt with similarly until a right input token is available. If a right input token is available it is (i) extracted, i.e. passed together with the left input token, and (ii) kept in a memory. Subsequent left input tokens are matched with the memory contents. Subsequent right input tokens are ignored.

EA-EXTRACT ABORT (success: put out both tokens
failure: put out a special EMPTY signal)

The dataflow net for the EA-matching function is shown in figure 2.8.4. When a left input arrives and a right input token is available, both inputs are extracted. If no right input is available, the left input is gated to a SINK-box and an EMPTY signal is emitted over the abort-output line.

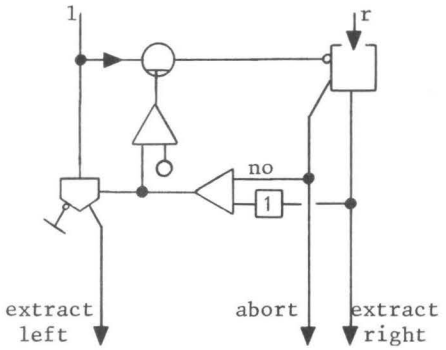


Figure 2.8.4. The EA-matching function.

PG: PRESERVE GENERATE (success: put out the left input and the memory token,
failure: put the left input token in the memory and
put out a special EMPTY token)

The dataflow net for the PG-matching function is shown in figure 2.8.5. When the first left input token arrives and there is a right input token available the right token is preserved, and both input tokens are extracted. If no right input is available, the left input is preserved and an EMPTY token is emitted over the abort line. Subsequent left input tokens are matched with the preserved value.

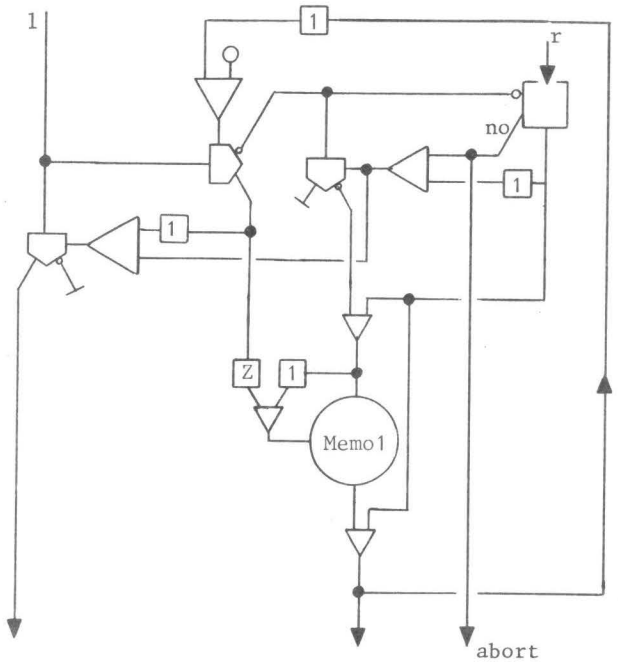


Figure 2.8.5. The PG-matching function.

In practice primitive building blocks such as memory cells and matching functions are realized as a piece of special hardware and not by a dataflow program. However, these results show the adequacy of our model, i.e., we can describe the meaning of the matching functions within the model of dataflow nets.

2.9. MODELLING PETRI-NETS

In this section we will show that Petri-nets can be modelled by our dataflow nets very naturally. Petri-nets are non-deterministic, and so to model this non-determinism we will build a random generator based on FM-subnets. Our definition of Petri-nets conforms to Peterson [69].

Definition 2.9.1. A *Petri-net* is a four-tuple (P,T,I,O) where

P is a set of *Places*,

T is a set of *Transitions*,

I is an input function I : T → Power(P),

O is an output function O : T → Power(P),

and where Power(P) is the set of all subsets of P.

The places can be *marked* with a number of *tokens*. Tokens do not have distinct values. A transition can *fire* if all its input places are marked. Firing means

removing one token from all input places and adding one token to all output places. An *execution* is a sequence of markings $\mu_0,\mu_1,\ldots$ The first marking μ_0 is called the *initial marking*. Every other marking μ_{i+1} is derived from its predecessor μ_i by the firing of one transition.
□

A Petri-net can be drawn as a bipartite directed graph with two types of nodes (drawn as ○ for places and | for transitions). If place p is in I(t) then there is an edge from p to t. If place p is in O(t) then there is an edge from t to p. As an example, figure 2.9.1. shows the graph representation of the Petri-net N defined as follows:

$$N = (\{P_1,P_2,P_3,P_4,P_5\},\{t_1,t_2,t_3,t_4\},\{t_1 \rightarrow \{P_1\},t_2 \rightarrow \{P_2,P_3,P_5\},t_3 \rightarrow \{P_3\},$$
$$t_4 \rightarrow \{P_4\}\},\{t_1 \rightarrow \{P_2,P_3,P_5\},t_2 \rightarrow \{P_5\},t_3 \rightarrow \{P_4\},t_4 \rightarrow \{P_2,P_3\}\})$$

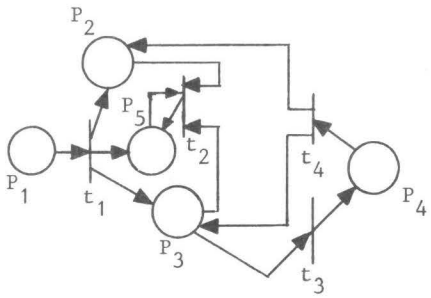


Figure 2.9.1. A Petri-net.

Dots in a place (◉) represent the marking of that place. The non-deterministic behaviour of Petri-nets is exemplified by two phenomena: *conflict* and *sharing*. Conflicting transitions have a common input place (figure 2.9.2.(a)). Either one of the transitions can fire if the place is marked. Two (or more) transitions can share a common output place (figure 2.9.2.(b)). The place is marked after firing of either one of the transitions.

Figure 2.9.2. Non-determinism in Petri-nets.

Theorem 2.9.1. *For every Petri-net* N *with initial marking* μ_0 *there is a dataflow net simulating it, i.e., for every execution of the Petri-net there is an equivalent execution of the dataflow net.*

Proof. We simulate a Petri-net N by mapping every transition with m inputs and n outputs to a dataflow subnet T(m,n) and by mapping every place with m inputs, n outputs and k initial tokens to a dataflow subnet P(m,n,k). The T and P subnets are then put together just as their counterparts in the graph representation of the Petri-net are.

(i) Construction of T(m,n).

T(m,n) must take in m inputs, one from each of its input lines and produce n outputs, one on each of its output lines. The construction of T(m,n) is therefore straightforward. It consists of an A(m,n) subnet defined below surrounded by a sluice construction. The sluice is needed here to prevent incomplete input tuples from passing and marking places that might not be marked in the corresponding Petri-net. If m=n, A(m,n) consists of m edges (figure 2.9.3.(a)). If m>n, m-n input lines are shut off by a SINK-box (figure 2.9.3.(b)). If m<n, n-m DUP-boxes are added (figure 2.9.3.(c)).

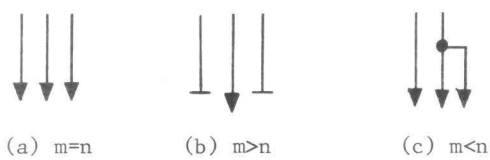


Figure 2.9.3. A(m,n) subnets.

(ii) Construction of P(m,n,k).

First we shall construct P(m,n,o) that simulates an initially empty place. A P(m,n,o) subnet must take in a token from any of its m inputs and send it to one of its output-lines chosen at random. This is accomplished by a *fan-in fan-out* construction as in figure 2.9.4.

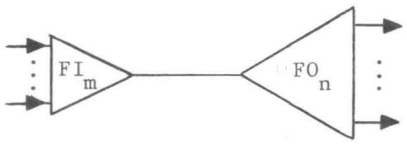


Figure 2.9.4. A P(m,n,o) subnet.

A fan-in subnet with m inputs and one output is just a tree of m-1 FM-subnets, with one exception when m=0 (see figure 2.9.5.).

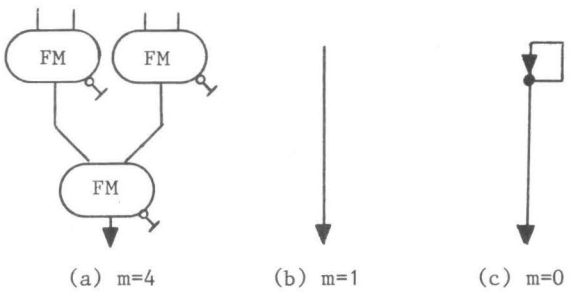


Figure 2.9.5. Fan-in subnets.

A fan-out subnet with one input and n outputs is the same as a T(1,n)-subnet, but with ANY-subnets instead of DUP-boxes. An ANY-subnet (see figure 2.9.6.) copies its input to either of its two outputs.

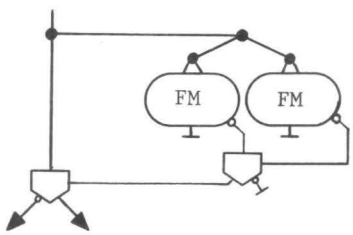


Figure 2.9.6. The ANY-subnet.

A P(m,n,k) net is just a P(m+k,n,o) net with k of its inputs connected to ZERO-boxes.

Now clearly, for every execution of N there is an execution of the dataflow net simulating it.

□

The contrary, though, happens not to be true: there are executions in the dataflow net for which there are no equivalent executions in N. This occurs, for example, when N contains a subnet as shown in figure 2.9.7.

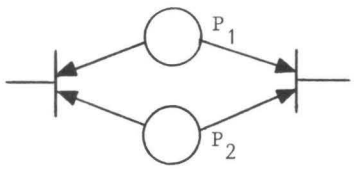


Figure 2.9.7. Petri subnet.

In a certain execution of the simulating dataflow net P_1 can send a token to the right while P_2 sends a token to the left.

CHAPTER THREE

THE DESIGN AND IMPLEMENTATION OF A HIGH LEVEL DATAFLOW LANGUAGE: DYNAMIC NETWORKS OF PROCESSES

3.1. INTRODUCTION

To express and analyse parallel algorithms we need a programming language based on a parallel model of computation. In our study this will be the model of data driven computation, where computing stations communicate with each other via channels, i.e., buffers of values operated in queue fashion. A program in our language will specify the topology of such a computation graph and the behaviour of the computing stations. We want this language to be powerful enough to serve as a problem solving tool yet simple enough to be elegantly defined and implemented. The following considerations were used as guidelines in the design of the language.

- Parallelism must be explicitly expressible.
- There must be a simple mapping from processes in our language to computing stations in the computation graph.
- The processes in our language must have the expressive power of procedures or modules (the dataflow processing elements from chapter two do not suffice).
- The computation graph must be adaptable to the problem size and data, i.e., we need a mechanism for dynamic process creation.
- There must be no need for global information about the computation graph when part of the graph is changing (because of dynamic process creation). The only communication must be via the edges of the graph. We call this the *locality principle*.
- The number of connections to "the outside world" on program level as well as on process level must be limited (not variable with the problem size) corresponding to physical reality.
- In the design we will concentrate on novel aspects, the choice of the rest of the language will be made such that it is easily implemented.

Parallelism inside processes will not be considered. A reasonable compiler can identify it and translate it for a given target machine. Also, if the target machine is a dataflow machine, processes will be translated into dataflow nets using dataflow analysis techniques as studied in ([85],[88],[3], [66]).

The programming language CSP [39] matches our requirements closely, except that CSP programs are static, i.e., they have a fixed computation graph. Moreover, CSP processes communicate with each other by name thus preventing most useful forms of dynamic process creation (knowing each others name is in fact a violation of the locality principle). The family of languages based on monitors and remote procedure calling (SIMULA, concurrent PASCAL, MODULA-2, DP) is even further away from our goals: the underlying computation graphs are again static, processes share data, and remote procedure calling violates the locality principle.

The simple language for parallel programming presented by Kahn [46] provides a good starting point for our language, and can be easily extended with dynamic process creation.

This chapter will introduce the language DNP (Dynamic Networks of Processes) based on Kahn's language. In section 3.2. we shall describe the language, and in section 3.3. we shall deal with an experimental implementation of it.

3.2. THE LANGUAGE DNP: DYNAMIC NETWORKS OF PROCESSES

DNP was implemented using a parser generator called PGEN [27]. Therefore, the syntax of DNP will be presented here in the format used by PGEN. In section 3.2.2. we will describe the *static part* of DNP, and in section 3.2.3. *the dynamic part*.

3.2.1. Syntax format.

The format is an extension of the familiar BNF-notation and figure 3.2.1.1. shows a self-definition of this format, taken from [27].

A non-terminal is enclosed by the brackets < and >. A terminal is either a keyword or a string. A keyword is a sequence of upper-case letters denoting the same sequence in lower case. A string is a non-empty sequence of characters surrounded by a single quotes. So the keyword BEGIN denotes the terminal symbol *begin*, and so does 'begin'. <id> stands for identifier.

```
<syntax>          ::= <rule>*.
<rule>            ::= <rule-name> '::=' <rule-body>'.'.
<rule-body>       ::= { <alternative> '|'}*.
<alternative>     ::= <primary>+.
<primary>         ::= (<terminal-symbol>|<rule-name>|<compound>)['+'|'*']
                      |<list>
                      |<option>.
<option>          ::= '['<rule-body>']'.
<list>            ::= '{'<primary> <terminal-symbol>'}' ('+'|'*').
<compound>        ::= '('<rule-body>')'.
<terminal-symbol> ::= <keyword>|<string>.
<rule-name>       ::= '<' <id> '>'.
```

Figure 3.2.1.1. The syntax format.

Using the terminology from figure 3.2.1.1., a syntax consists of a sequence of rules, where each rule is a non-terminal followed by ::= followed by a series of alternatives separated by vertical bars |. An <option> indicates that one of the enclosed alternatives may not occur. An asterisk * indicates zero or more repetitions of some notion; a plus-sign + indicates one or more repetitions. A <compound> groups a structure into a notion. A <list> is a sequence of notions separated but not terminated by a terminal symbol. So

```
( <id> ',')*
```

stands for zero or more identifiers followed by a comma, such as

```
a,b,c,
```

while

```
{ <id> ','}*
```

stands for zero or more identifiers separated by commas, such as

a,b,c

We will comment on the use of PGEN in section 3.3.1.

3.2.2. DNP - static part.

A DNP program consists of a number of process declarations and a main body.

syntax:

```
<dnp-program>      ::= <process-decl>* <main>.
```

In the main body processes are activated. They are connected together and to the outside world by channels, which are queues of tokens or values. For every channel there is one producing process and one consuming process. A process declaration consists of a heading and a body. In the heading formal channels are declared, specifying whether the channel is an input channel or an output channel. A process heading must contain at least one formal channel. Apart from formal channels, formal value parameters can also occur in the heading.

syntax:

```
<process-decl>     ::= <process-heading> ':' <process-body>.
<process-heading>  ::= PROCESS <id> '(' <channels> [<values>]')'.
<channels>         ::= ( <inchannels>|<outchannels> )+.
<inchannels>       ::= IN {<id> ','}+.
<outchannels>      ::= OUT {<id> ','}+.
<values>           ::= (<type> {id ','}+) +.
```

where <type> is a type declaration such as *int* or *char*.

The body of a process declaration consists of three types of components:

(i) internal statements and declarations
(ii) communication statements
(iii) expansion statements.

Internal statements and declarations are ordinary statements (condition, loop, internal data declaration) that only change the internal state of the process. They could have been borrowed from any programming language, and in our case were borrowed from C [49], the UNIX system implementation language.

Communication statements allow a process to read (consume a value from an input channel) and write (produce a value on an output channel). If a channel is empty when the consumer process performs a read on it, the consumer process is blocked until the producer process has written a value on the channel. The communication statements are in fact implemented as ordinary C-functions, supplied in the run-time environment. There is therefore no syntactic difference between internal statements, declarations and communication statements: they all look like C.

syntax:

<process-body> ::= BEGIN (<expansion>|<c>)* END.
where <c> stands for a piece of C program text inside a C function declaration.

A main body declaration has the same structure as a process declaration. The input and output channels in its heading are the input and output files connecting the program to the outside world, and in the body the initial computation graph is set up by naming the internal channels and processes in an expand statement causing the main body to create processes and connecting them by channels. (A dynamic version of expansion where the network can be changed while executing will be introduced in section 3.2.3.)

syntax:

```
<main>          ::= MAIN <id> '(' <channels> ')'':'
                    BEGIN   [<c>]
                            EXPAND [CHAN {<id> ','}+]
                                   <creation>+
                            ENDEXP
                            [<c>]
                    END.
<creation>      ::= CREATE  <id> '(' <channels> [<values>] ')'.
```

The CHAN part declares the internal channels. The create statements initiate processes with either internal channels or the channels of the main body as actual channel parameters. Every input channel of the main body will occur once as an actual input channel in a creation, just as every output channel of the main body will occur once as an actual output channel. The internal channels will occur twice, once as an actual input channel and once as an actual output channel of distinct processes. This gives us a well-formed graph (every internal channel being an edge), connected to the environment by the input and output channels of the main body.

We have now defined the static part of DNP and will illustrate it by an example: this program will produce the integers $2^i 3^j$ in ascending order on an output channel (see figure 3.2.2.1.).

```
process times(in i out o int f) :
begin int v;
        while (read_int(i,&v)) write_int(o,f*v);
end
process order(in i2,i3 out m) :
begin int v2,v3;
        read_int(i2,&v2); read_int(i3,&v3);
        do { if(v2<v3) {write_int(m,v2); read_int(i2,&v2);}
             else  if (v3<v2)
                        {write_int(m,v3); read_int(i3,&v3);}
                   else
                        {write_int(m,v2);
                         read_int(i2,&v2);
                         read_int(i3,&v3);
                        }
           }
        while (1);
end
process triplicate(in m out o1,o2,o3 int init) :
begin int v = init;
        while(write_int(o1,v),write_int(o2,v),write_int(o3,v))
                read_int(m,&v);
end
main Hamming(out f23) :
begin int one = 1, two = 2,  three = 3;
      expand chan m,i2,i3,o2,o3
             create triplicate(in m out f23,i2,i3 int one)
             create times(in i2 out o2 int two)
             create times(in i3 out o3 int three)
             create order(in o2,o3 out m)
      endexp
end
```

Figure 3.2.2.1. A static DNP program.

This program is connected to the outside world by the output channel *f23*. Figure 3.2.2.2. shows the computation graph of program Hamming.

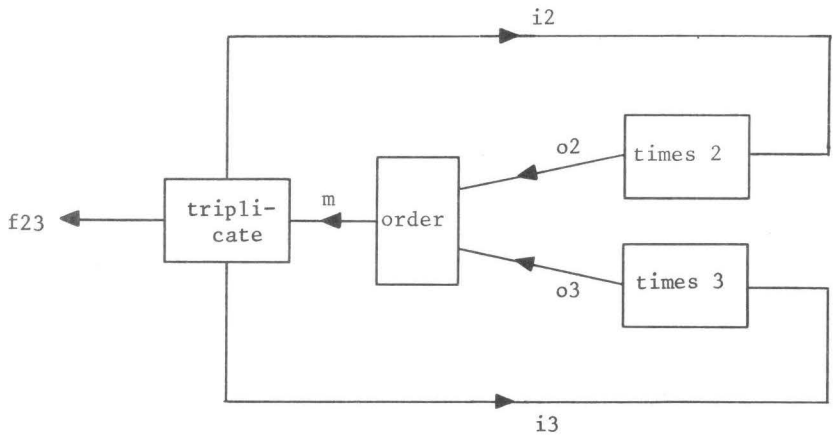


Figure 3.2.2.2. Computation graph of program Hamming.

3.2.3. DNP - dynamic part.

A process can replace itself by a subgraph (subnetwork) of processes by performing an expansion. The newly created subgraph is connected to the rest of the graph by the same channels as the old process was. An expand statement consists of a declaration of the new internal channels and a number of process activations. A process activation is either a process creation, i.e., a new process that starts in its initial state, or a survival. In a survival, the old process that caused the expansion is resumed possibly with different actual channels. Survival provides a way of inheriting the process state (data and control environment). At most one survival is allowed in an expansion.

syntax:

```
<expansion>      ::= EXPAND [CHAN {<id> ','}+]
                        (<creation>+ [<survival>]
                        |<survival>)
                    ENDEXP.
<survival>       ::= KEEP   <id> '(' <channels> ')'.
```

Notice the similarity between an expansion and the declaration of the initial graph in the main body. The newly created internal channels will occur twice, once as an input channel and once as an output channel. The old formal channels will occur once and their type (input or output) will not change. When an expansion is performed, the following takes place:

- the old process is disconnected from the network; its channels are temporarily closed,
- for every <creation> a new process is created,
- the newly created processes, and the old process if a survival occurred, are connected into a subnetwork by means of the internal channels,
- the subnetwork is connected to the rest of the graph by the temporarily closed channels,
- the new processes start computing in their initial state and, if it is still part of the subnetwork, the old process proceeds after the expand statement.

The rest of the network can carry on computing while the expansion takes place. Consider the following process declaration:

```
process compile (in source out object):
   begin .
      expand chan  e1,e2,e3,e4,e5
             create lex (in source out e1,e2)
             create scan1 (in e1 out e3,e4)
             create scan2 (in e2,e4 out e5)
             create codegen (in e3,e5 out object)
      endexp
   end
```

The expansion in this process declaration can be pictured as in figure 3.2.3.1.

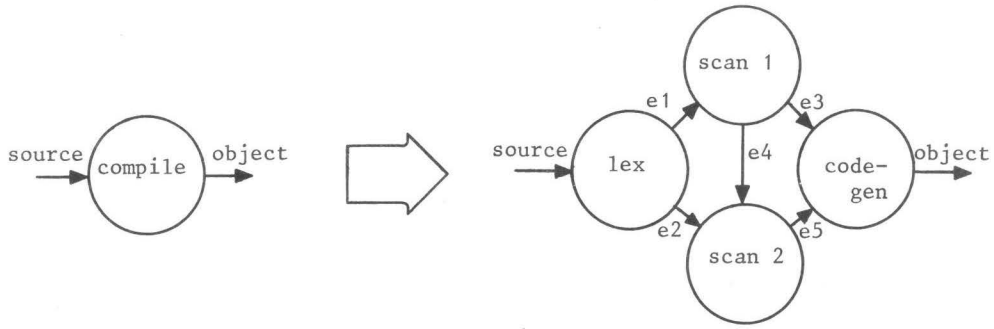


Figure 3.2.3.1. Expansion into a subnetwork.

An example of a dynamic DNP program is given in figure 3.2.3.2. It is a parallel version of the prime sieve of Eratosthenes. This example was inspired by an example given by McIlroy [61] for demonstrating the use of coroutines.

```
process filter(in ints out primes int factor):
begin   int i;
        read_int(ints,&i);
        while(i>0)
                {if((i % factor) != 0) write_int(primes,i);
                 read_int(ints,&i);
                };
        write_int(primes,-1);
end

process primesv(in factors out primes):
begin   int i;
        read_int(factors,&i);
        while(i>0)
                {expand chan inter
                        create filter(in factors out inter int i)
                        keep primesv(in inter out primes)
                 endexp
                 write_int(primes,i);
                 read_int(factors,&i);
                };
end

process ints(out o):
begin   int i;
        for (i=2; i<80; i++) write_int(o,i);
        write_int(o,-1);
end


main  Eratosthenes(out primes):
begin   expand chan inter1
                create ints(out inter1)
                create primesv(in inter1 out primes)
        endexp
end
```

Figure 3.2.3.2. A dynamic DNP program.

3.3. AN EXPERIMENTAL IMPLEMENTATION OF DNP

3.3.1. Introduction.

To implement a parallel language one needs a parallel machine, real or virtual. The UNIX operating system [72] is a parallel machine, with so called *pipes* for interprocess communication, *forking* for dynamic process creation, and with C as its machine language. The reason that C was chosen for the internal DNP statements, was that the task of implementing DNP was made easier, since only a preprocessor for C is needed. Figure 3.3.1.1. shows DNP features and their UNIX/C counterparts.

DNP	UNIX/C
channel	pipe/file
process declaration	C function declaration
process	process
creation	forking
internal statements in C	same C statements

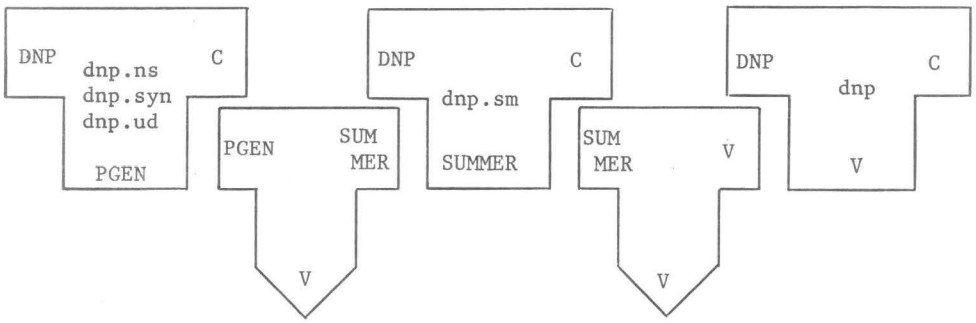
Figure 3.3.1.1. DNP features and their UNIX counterparts.

In a DNP program there is no limit to the total number of processes and channels, to the number of channels connected to one process, nor to the size of a channel, where the size of a channel is the number of values written but not yet read. In UNIX, unfortunately, there is a limit to all these values. We call our implementation experimental because we have chosen to live with these system limits, even though some of them, e.g. the maximum number of processes, are rather severe. Care has been taken to implement DNP so that a maximal number of DNP processes can be created by not wasting UNIX processes. We will come back to this when we discuss the translation of the expand statement.

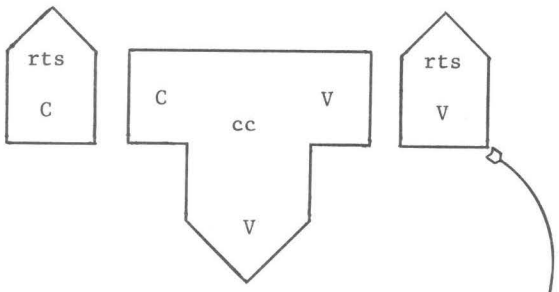
The DNP compiler was implemented using the parser generator PGEN [27], constructed at the Mathematical Centre, Amsterdam. As PGEN accepts only LL(1) grammars it was necessary to express DNP in that form, but this caused no

particular problems. The virtue of PGEN is that it automatically generates error messages in terms of the syntactic notions. The semantic actions must be written in SUMMER [51], a language well suited for that purpose. The facilities for communication between parser and semantic actions and between various semantic actions are unfortunately rather poor in PGEN. This kind of communication should proceed via derived and inherited attributes ([53],[54]). Only a very simple kind of derived attributes is implemented in PGEN: a notion or action is allowed to return one value. For the rest the compiler writer is forced to resign to the use of global variables. A revised implementation of PGEN with better communication facilities seems worth while because apart from this shortcoming PGEN is pleasant to work with.

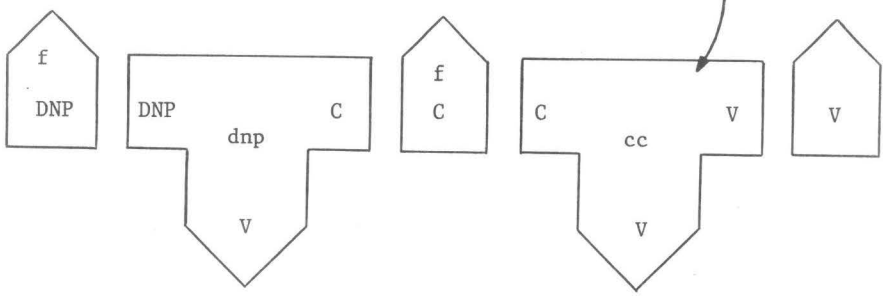
The compiler is, according to the rules of PGEN, structured as a lexical scanner (*dnp.ns*), a parser extended with semantic actions (*dnp.syn*), and a file containing global variables and procedures (*dnp.ud*). Figure 3.3.1.2. shows the various parts of the DNP-system in terms of T-diagrams [25], where V stands for a computer or its machine language, rts for a runtime system, and f for a user program. Running a compiled DNP-program (the result of 3.3.1.2.(c)) involves linking it with the run-time system (the result of 3.3.1.2.(b)).



(a) DNP-compiler generation



(b) Run-time system generation



(c) DNP-compilation.

Figure 3.3.1.2. The DNP-system.

3.3.2. The translation of DNP to C.

The DNP compiler is a preprocessor that translates DNP into C. Every DNP-process is translated to a C-function and a DNP-main body is translated to a C-main procedure. Channels connecting the network to the outside world are implemented by files. Channels connecting processes to each other are implemented by pipes. A pipe is a communication buffer between UNIX-processes represented by a read-file-descriptor and a write-file-descriptor. The compiler will ensure that only one process, viz. the consumer process, will control the read-file descriptor and only one process, viz. the producer process, will control the write-file-descriptor.

A process-heading is translated into a C-function heading, and the relevant information about formal input and output channels is kept in some global variables.

A process-body is a sequence of <c>-s and <expansion>-s. The lexical scanner collects all C-text between a BEGIN and an <expansion>, or an <expansion> and an <expansion>, or an <expansion> and an END, and yields it as one lexical symbol to the parser. The parser just outputs this piece of C-text. Errors in the C-text will be detected by the C-compiler. An <expansion> will be translated into a C compound statement.

When an <expansion> is encountered the compiler checks whether the formal and internal channels are used properly. If so, it generates code

(1) to allocate pipes for the internal channels,
(2) to allocate processes for all activations except the last one,
(3) to make the appropriate process-channel connections,
(4) to start the processes with the right formal/actual channel-identifications.

For the last activation, whether a creation or survival, no process needs to be allocated, because the process that performs the expansion can be used for it. This trick saves one UNIX-process per expansion, but makes the code-generation process more complex. The last activation must be handled

differently but, because the parser is based on the LL(1) recursive descent technique, it only knows that a particular activation is the last one after it has been parsed completely. We therefore generate code for an activation when we encounter its successor, or we encounter the ENDEXP symbol.

Pipe and process allocation are implemented by standard UNIX system calls (pipe and fork). A new process is an exact copy of the process that produced it, except for an integer returned by the fork operation. Because a new process is an excact copy of the old one, all pipes and files available to the old process are available to the new one via their descriptor. It is therefore necessary for a process to close the files and pipes it does not need.

Starting a creation is implemented by a function call. Starting a survival is implemented by a number of channel assignments. As a survival is the last activation of an expansion, control will pass automatically to the correct instruction.

In order to make the above description more concrete we will consider the translation of the process of figure 3.3.2.1.

```
process T (in i out o):
   begin
      if (test)
         expand chan l1,l2,l3,l4
                create T (in l1 out l2)
                create T (in L3 out l4)
                create N (in i, l2,l4, out o,l1,l3)
         endexp
   end
```

Figure 3.3.2.1. Example process.

The heading of DNP-process T is translated such that there is a UNIX-process where i is identified with a read-file-descriptor and o is identified with a write-file-descriptor. This UNIX process will execute the C-function T(i,o) as pictured in figure 3.3.2.2.

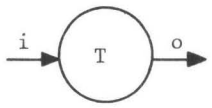


Figure 3.3.2.2. T(i,o).

Upon encountering *expand chan l1,l2,l3,l4* code could be generated to allocate four pipes. This is not done because:

(i) UNIX allows a rather small number of open files (a pipe counts for two files) per process, and

(ii) as a process is only allowed to control a subset of all the pipes, most of these will have to be closed afterwards.

Therefore code is generated to allocate a pipe only when it is really needed.

create T (in l1 out l2) will be checked for correct use of channels, and will be translated to:

(1) allocate two pipes 11 and 12,

(2) create a new process (by means of a *fork* statement). Now there are two processes, a *parent* and a *child*. Both processes control pipes 11 and 12, and files i and o,

(3) the child will perform *T(in l1 out l2)* and will therefore close the write-file-descriptor of 11, the read-file-descriptor of 12 and the files i and o,
the parent closes the read-file-descriptor of 11 and the write-file-descriptor of 12,

(4) the child calls *T(l1,l2)*,
the parent goes on with the expansion.

These steps are picture in figure 3.3.2.3., where a pipe is an arrow ▢⟶ with the front part ▷ its read-file and the back ▢ its write-file.

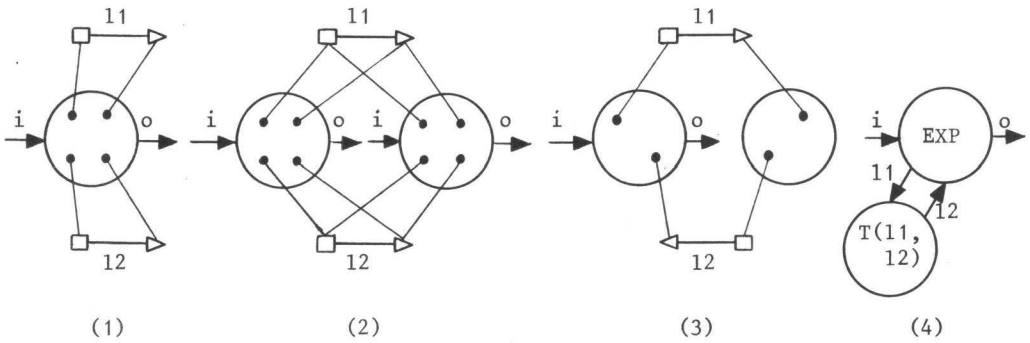


Figure 3.3.2.3. Steps in a process activation.

create T(in l3 out l4) will be translated similarly. For the last process activation *create N(in i,l3,l4 out o,l1,l2)* no new process is needed. It is translated to a function *N(i,l3,l4,o,l1,l2)*. The end of a process declaration is translated to

(1) write end of information on all output files,
(2) read all input files until end of information,
(3) exit.

Figure 3.3.2.4. shows the C translation of the example program Erathostenes from figure 3.2.3.2.

```
#include "rts.h"
filter(ints,primes,factor)
inchan ints;
outchan primes;
int factor;
{
        int i;
        read_int(ints,&i);
        while(i>0)
        {
                if((i % factor) != 0) write_int(primes,i);
                read_int(ints,&i);
        };
        write_int(primes,-1);

        putc(primes,EOF);
        while(getc(ints)!=EOF);
        exit(0);
}
primesv(factors,primes)
inchan factors;
outchan primes;
{
        int i;
        read_int(factors,&i);
        while(i>0)
        {
                {
                        struct channel inter;
                        connection(&inter);
                        if((_f=fork())==-1) error("Cannot create process");
                        if(_f==0){ /*  son  */
                                close(primes);
                                close(inter.i);
                                filter(factors,inter.o,i);
                        }
                        close(factors);
                        close(inter.o);
                        init_queue(&_m);
                        ins_q(&_m,inter.i);
                        ins_q(&_m,primes);
                        factors = del_q(&_m);
                        primes = del_q(&_m);
                }

                write_int(primes,i);
                read_int(factors,&i);
        };

        putc(primes,EOF);
        while(getc(factors)!=EOF);
        exit(0);
}
```

```
ints(o)
outchan o;
{
        int i;
        for (i=2; i<80; i++) write_int(o,i);
        write_int(o,-1);

        putc(o,EOF);
        exit(0);
}
main()
{
        int primes;
        primes=creat("primes",0666);
        {
                {
                        struct channel inter1;
                        connection(&inter1);
                        if((_f=fork())==-1) error("Cannot create process");
                        if(_f==0){ /*  son  */
                                close(primes);
                                close(inter1.i);
                                ints(inter1.o);
                        }
                        close(inter1.o);
                        primesv(inter1.i,primes);
                }
        }
        exit(0);
}
```

Figure 3.3.2.4. C translation of Eratosthenes.

3.3.3. Appendix: the compiler and the run-time system.

DNP was implemented on a VAX 11/780 running Berkeley UNIX 4.1, using PGEN [27] and Summer [51]. It consists of dnp.ud (user definitions), dnp.ns (a lexical scanner) and dnp.syn (the compiler). The run-time system is written in C.

```
#dnp.ns#

#-------------------------------------------------------------------------------
Communication between the parser, generated from dnp.syn
and this lexical analyser proceeds via global variables:
        sy
        t-sy
        lnr
        keytab
        predef
        kartab.
The procedure nextsym yields the input lexical symbols as
declared in dnp.syn. It also keeps track of linenumbers
in lnr and signals end of file EOF. A next lexical symbol
is put in sy and its type is put in t-sy. For further
details see the PGEN defining MC-report:

G. Florijn & G. Rolf
PGEN - A general purpose Parser Generator
MC IW157/81 januari 1981

-------------------------------------------------------------------------------#




const STATE_C := 0,
      STATE_DNP := 1;

var letter := upper || lower,
    alpha := letter || digit,
    true  := 1,
    layout := ' \t',
    empty  := '';

var state := STATE_DNP,
    infile := stand_in;

proc ermsg(msg, lino) #print errormessage on standard error output #
(stand_er.put('error near line ', lino, ': ', msg, '\n')
);
```

```
#-------------------------------------------------------------------------------
proc nextsym is either in STATE_C or in STATE_DNP
when in STATE_C it yields:
        - EOF if there is no more input
        - all C-text until the next DNP keyword (and goes in STATE_DNP)
                if there is C-text on input
        - the next DNP symbol if there is no C-text (and it goes into STATE_DNP)
when in STATE_DNP it yields:
        - EOF if there is no more input
        - the next DNP symbol
          if the symbol is begin or endexp it goes into STATE_C
-------------------------------------------------------------------------------#
proc nextsym()
( case state of
  STATE_C:
    var pre, kw;
    sy := '';
    while true
    do scan line
       for
         if pre := break(letter)
         then
            sy := sy || pre;
            kw := span(alpha);
            if kw = 'end' | kw = 'expand'
            then
               state := STATE_DNP;
               if scan sy for (span(' \t\n') | lit('')) & rpos(0) rof
               then
                  sy := kw;
                  t_sy := keytab[sy];
                  return;
               else
                  move(-kw.size);
                  t_sy := predef['c_text'];
                  return
               fi
            else
              sy := sy || kw
            fi
         else
            sy := sy || line.rtab(0) || '\n';
            if line := scan_string(infile.get()) fails
            then
               sy := 'EOF'; t_sy := predef['EOF']; return
            else
               lnr := lnr + 1
            fi
         fi
       rof
     od,
```

```
  STATE_DNP:
    while true
    do line.span(layout) | empty;
       if sy := line.any(letter)
       then
          sy := sy || (line.span(alpha) | empty);
          if keytab[sy] ~= undefined
          then
             t_sy := keytab[sy]
          else
             t_sy := predef['id']
          fi;
          if sy = 'begin' | sy = 'endexp'
          then
             state := STATE_C
          fi;
          return
       elif sy := line.move(1)
       then
            t_sy := kartab[sy];
            return
       else
            if line := scan_string(infile.get()) fails
            then
               sy := 'EOF';
               t_sy := predef['EOF'];
               return;
            else
               lnr := lnr + 1
            fi;
       fi
    od;
  esac
);
```

```
# dnp.ud #


var
in_formals,       # formal input channels of a process declaration
                    built up while parsing <process-heading> or <main>
                    read by <expansion>, <survival>, <process-body>
                  #

out_formals,      # same for  formal output channels #

free_in_formals, #unused formal input channels in an expansion
                   initially equal to in_formals
                   emptied successively by /checkin/ actions
                   checked by <expansion>.EXIT
                  #

free_out_formals,#same for formal output channels in an expansion #

intermediates,    # the new intermediate channels in an expansion
                    for an intermediate we must generate code to create
                    a pipe, which is done the first time the intermediate
                    is used as an actual input or output channel
                  #


free_in_parts,    # unused input parts of intermediates during parsing
                    of an expansion
                    checked and emptied by chechin actions
                  #

free_out_parts,   # same for unused output parts of intermediates  #

to_close,         # open files: formal channels, input parts of intermediates,
                              output parts of intermediates.
                    some files are already created, because for complete channels
                    a pipe is created, but are not needed in a certain process.
                    these files must be closed
                    when a pipe is created for an intermediate x, x.i and x.o
                    are added to to_close in checkin or checkout actions.
                    when x.i is used in a process, it is removed from to_close
                  #

procname,         # process name in a creation or survival #

curproc,          # current process declaration #
```

```
proctab := table(20,''),
                # key: process name
                  entry: io-channel-pattern
                  used for checking consistency of def and uses of
                  a process by checkchan_ud
                #

firsttab := table(20,0),
                # key: process name
                  entry: line number first occurence
                  used for error msg by checkchan_ud
                #


gen_call,       # this one is needed because we cannot see when parsing
                  a creation or survival that it is the last one. So
                  code generation for creations will happen when the
                  next creation or survival or expend is encountered.


                  we have two cases:
                  (1) a creation x which is not the last process
                      determined when encountering successor of x, only
                      if x exists (gen_call = TRUE).
                      this happens in init_ud
                  (2) x is the last process
                      (2.1) creation (gen_call = TRUE): generate function call
                            this happens in <expansion>.EXIT
                      (2.2) survival: generate channel part assignments
                            this happens in <survival>.EXIT
                #

actualparts,    # actual channel parts of of creation or survival #

actualvals,     # same for actual value parameters #

pre,
post,
rest;           # used for pattern matching #

proc match(str,pat) #pattern matching #
(
return(
       scan str for pre := find(pat) & lit(pat) & post := rtab(0)
                rof  &
                    rest := pre || post
      )
)
;
```

```
# init_ud generates code for previously parsed creation, if any (gen_call)
          initializes global variables for the new creation or survival
#
proc init_ud()
(var actuals;
        if gen_call = 'TRUE'
        then  put('if((_f=fork())==-1) error("Cannot create process");\n');
              put('if(_f==0){ /*  son  */\n');
              scan to_close
               for move(1);
                   while pre := find(',')
                      do move(1);
                         put('close('  ,pre,  ');\n');
                      od
               rof;
              actuals := actualparts || actualvals;
              put(procname,'(',actuals.substr(0,actuals.size-1),');\n');
              put('}\n');
              scan actualparts
               for while pre := find(',')
                      do move(1);
                         put('close('  ,pre,  ');\n');
                      od
               rof
        fi;
        actualparts := ''; actualvals := '';
        gen_call := 'TRUE';
)
;


#checkchan_ud checks consistency of def and use of channels in process
              declaration, creation and survival
#
proc checkchan_ud(pnm,iopat)
(       if proctab[pnm] = ''
        then proctab[pnm] := iopat; firsttab[pnm] := lnr
        else if proctab[pnm] ~= iopat
             then ermsg('channels inconsistent with line ' ||
                        string(firsttab[pnm]),lnr
                       )
             fi
        fi
)
;
```

```
#checkin_ud generates pipe creation code (connection) if needed
            checks correct use of input channel
#
proc checkin_ud(nm)
(       scan intermediates #if channel used first generate "connection"#
            for  if   pre := find(',' || nm || ',')
                 then lit(','||nm); intermediates := pre || rtab(0);
                      put('connection(&'  ,nm,  ');\n');
                      to_close := to_close || nm || '.i,' || nm || '.o,';
                 fi
            rof;




            if scan free_in_parts
               for pre := find(',' || nm || ',')
                   &
                   (lit(',' || nm); free_in_parts := pre || rtab(0);
                     scan to_close
                      for if pre := find(','||nm||'.i,')
                       then lit(','||nm||'.i');
                            to_close := pre || rtab(0);

                          fi

                     rof;
                     actualparts := actualparts || nm || '.i,'
                    )
               rof fails
            then
             if scan free_in_formals
               for pre := find(',' || nm || ',')
                   &
                   (lit(',' || nm); free_in_formals := pre || rtab(0);
                    scan  to_close
                     for if pre := find(','||nm||',')
                       then lit(','||nm);
                            to_close := pre || rtab(0);
                          fi
                     rof;
                    actualparts := actualparts || nm || ','
                   )

               rof fails
             then ermsg('wrong input channel '|| nm, lnr)
             fi
            fi;
)
;
```

```
proc checkout_ud(nm) # see comment checkin_ud #
(       scan intermediates
            for  if pre := find(',' || nm || ',')
                 then lit(','||nm); intermediates := pre||rtab(0);
                      put('connection(&'  ,nm,  ');\n');
                      to_close := to_close || nm || '.i,' || nm || '.o,'
                 fi
            rof;
            if scan free_out_parts
               for pre := find(',' || nm || ',')
                   &
                   (lit(',' || nm); free_out_parts := pre || rtab(0);

                    scan to_close
                     for if pre := find(','||nm||'.o,')
                       then lit(','||nm||'.o');
                            to_close := pre || rtab(0);
                          fi

                     rof;

                    actualparts := actualparts || nm || '.o,'
                   )
               rof fails
            then
             if scan free_out_formals
               for pre := find(',' || nm || ',')
                   &
                   (lit(',' || nm); free_out_formals := pre || rtab(0);

                    scan to_close
                     for if pre := find(','||nm||',')
                       then lit(','||nm);
                            to_close := pre || rtab(0);
                          fi
                     rof;

                    actualparts := actualparts || nm || ','
                   )

               rof fails
             then ermsg('wrong output channel '|| nm, lnr)
             fi
            fi;

)
;
```

```
# dnp.syn #

LEXICAL id, c-text.



<dnp-program> ::= <process-decl>* <main> .

INIT:   put('#include "rts.h"\n');



<process-decl> ::= <process-heading> ':' <process-body> .



<process-body> ::= BEGIN
                        (<expansion> | <c>)*
                   END.

INIT:   put('{\n');


EXIT:
        while  match(out_formals, ',')
                do put('putc(' ,pre, ',EOF);\n'  ); out_formals := post ; od;
        while  match(in_formals, ',')
                do
                  put('while(getc(' ,pre,  ')!=EOF);\n');
                  in_formals := post;
                od;
        put('exit(0);\n}\n');



<c> ::= t : <c-text> .

EXIT:   put(t, '\n');
```

```
<process-heading> ::= PROCESS pid: <id>  /procname/
        '(' ( ( IN { nm: <id> /inname/ ',' }+ )
            | ( OUT { nm: <id> /outname/ ',' }+ )
            )+
              (t: <type> {nm: <id> /valname/ ','}+ )*
        ')' .

INIT:
        var val_formals := '', c_pack := '' , iopat := '';
        in_formals := ''; out_formals := '';


/procname/: curproc := pid;


/inname/:
        in_formals := in_formals || nm || ',';
        c_pack := c_pack || nm  || ',';
        iopat := iopat || 'i';


/outname/:
        out_formals := out_formals || nm || ',';
        c_pack := c_pack || nm || ',';
        iopat := iopat || 'o';


/valname/:
        val_formals := val_formals || t || ' ' || nm || ';\n';
        c_pack := c_pack || nm || ',';


EXIT:   checkchan_ud(pid,iopat);
        put(pid, '(');
        if c_pack ~= '' then put(c_pack.substr(0,c_pack.size-1)) fi;
        put(')\n');
        if in_formals ~= ''
        then put('inchan ');
             put(in_formals.substr(0,in_formals.size-1));
             put(';\n')
        fi;
        if out_formals ~= '' then put('outchan ');
                        put(out_formals.substr(0,out_formals.size-1));
                        put(';\n')
        fi;
        if val_formals ~= '' then put(val_formals) fi;
```

```
<type> ::= INT /i/ | CHAR /c/ .

INIT:   var kw;

/i/:    kw := 'int';

/c/:    kw := 'char';

EXIT:   return(kw);


<expansion> ::=
        EXPAND
           [ CHAN {nm: <id> /chname/ ','}+  ]
           /chdecl/
           <creation>* [<survival>]
        ENDEXP .

INIT:   var m,actuals;
        free_in_formals := ',' || in_formals;
        free_out_formals := ',' || out_formals;
        intermediates := ',';
        gen_call := 'FALSE';
        to_close := ',' || in_formals || out_formals;
        put('{\n');

/chname/: intermediates := intermediates || nm || ',';


/chdecl/: if intermediates ~= ','
          then put('struct channel ',
                   intermediates.substr(1,intermediates.size-2),
                   ';\n'
                  )
          fi;
          free_in_parts := intermediates; free_out_parts := intermediates;

EXIT:   m := free_in_formals || free_out_formals ||
             free_in_parts || free_out_parts;
        if m ~= ',,,,'
        then while match(m,',') do m := pre || ' ' || post od;
             ermsg('unused channel(s) '|| m, lnr)
        fi;

        if gen_call = 'TRUE'
        then actuals := actualparts || actualvals;
             put(procname,'(',actuals.substr(0,actuals.size-1),');\n')
        fi;
        actualparts := ''; actualvals := '';
        put('}\n');
```

```
<creation> ::=
        CREATE pid: <id>    /processname/
        '(' ( ( IN  {nm: <id> /checkin/ ','}+ )
            | ( OUT {nm: <id> /checkout/ ','}+ )
            )+
              ( <type> {nm: <id> /addval/ ','}+ )*
        ')' .

INIT:   var iopat := '';
        init_ud();


/processname/:  procname := pid;

/checkin/: checkin_ud(nm);
           iopat := iopat || 'i';

/checkout/: checkout_ud(nm);
            iopat := iopat || 'o';


/addval/ : actualvals := actualvals || nm || ',';


EXIT:   checkchan_ud(pid,iopat);
```

```
<survival> ::=
        KEEP pid: <id> /procname/
        '(' ( ( IN {nm: <id> /checkin/ ','}+ )
            | ( OUT {nm: <id> /checkout/ ','}+ )
            )+
        ')' .
INIT:   var formals, iopat := '';
        init_ud();

/procname/: if pid ~= curproc
            then ermsg('incorrect process in survival', lnr)
            fi;

/checkin/:  checkin_ud(nm);
            iopat := iopat || 'i';

/checkout/: checkout_ud(nm);
            iopat := iopat || 'o';
```

```
EXIT: checkchan_ud(pid,iopat);
      formals := in_formals||out_formals;
        put('init_queue(&_m);\n');
        while match(actualparts,',')
                do      actualparts := post;
                        put('ins_q(&_m,',pre,');\n')
                od;
        while match(formals,',')
                do      formals := post;
                        put(pre,' = del_q(&_m);\n')
                od;
      gen_call := 'FALSE';
```

```
<main> ::=
        MAIN <id>
        '(' ( ( IN { nm: <id> /inname/ ',' }+ )
            | ( OUT { nm: <id> /outname/ ',' }+ )
            )+
        ')' ':'  /head/
        BEGIN /open/
              [<c>]
              <expansion>
              [<c>]
              /close/
        END .

INIT:   var files;
        in_formals := ''; out_formals := '';

/inname/: in_formals := in_formals || nm || ',';

/outname/: out_formals := out_formals || nm || ',';

/head/: put('main()\n{');
        files := in_formals || out_formals;
        put('int ', files.substr(0,files.size-1), ';\n');
        files := in_formals;
        while match(files,',')
          do put(pre, '=open("', pre, '",0);\n');
             files := post;
          od;
        files := out_formals;
        while match(files,',')
          do put(pre, '=creat("', pre, '",0666);\n');
             files := post;
          od;

/open/: put('{\n');

/close/: put('}\n');

EXIT:   put('exit(0);\n}\n');
```

```
/*rts.c */

#include "rts.h"


/* ERROR MESSAGE */
/* ============= */

error(msg) char *msg;
{ printf("ERROR: %s\n", msg); exit(0); }



/* CHANNEL CREATION */
/* ================ */

connection(ch) struct channel *ch;
{ int fildes[2];
  if(pipe(fildes) != 0)
        error("Cannot create pipe");
  ch->i = (inchan) fildes[0];
  ch->o = (outchan) fildes[1];
}



/* INPUT OUTPUT */
/* ============ */

putc(f,c)
int f;
char c;
{ write(f,&c,1); }

char getc(f)
int f;
{ char c;
  if (read(f,&c,1) != 1)
        return(EOF);
  else  return(c);
}
```

```
int read_int(from,val)
inchan from;
int     *val;
{ int fdfrom = (int) from, res = 0, sign = 1,h = *val;
  char c;
  *val = 0;
  do { c = getc(fdfrom);
     } while (!isdigit(c) && (c != '-') && (c != EOF));
  if(c == '-')
        { sign = -1;
          c = getc(fdfrom);
        }
  while (isdigit(c))
        { res = 1;
          *val = *val * 10 + c - '0';
          c = getc(fdfrom);
        }
  *val = sign * *val;
  if (!res) *val = h;
        return(res);
}

write_int(to,val)
outchan to;
int val;
{ int fdto = (int) to;
  if(val < 0)
        { putc(fdto,'-'); val = -val; }
  wint(fdto, val);
  putc(fdto, ' ');
}


wint(fdto, val)
int fdto, val;
{ if (val <= 9) { putc(fdto, '0' + val); }
  else { wint(fdto, val / 10);
         putc(fdto, '0' + val % 10);
       }
}
```

```
read_item(from)   /* an int, SEP or EOM */
inchan from;
{ int fdfrom = (int) from, res = 0;
  char c;
  while ((c=getc(from))!='*' && c!='$' && isdigit(c)==0);
  while (isdigit(c))
        { res = res * 10 + c - '0';
          c = getc(fdfrom);
        }
  if (c == '*')  res = EOM;
  else if (c == '$') res = SEP;
  return(res);
}

write_item(to,val)  /* an int, SEP or EOM */
outchan to; int val;
{ int fdto = (int) to;
  char c;
  if (val == EOM)
        { putc(fdto, '*'); putc(fdto, '\n'); return; }
  else if (val == SEP)
        { putc(fdto,'$'); putc(fdto,'\n'); return; }
  wint(fdto, val);
  putc(fdto, ' ');
}

isdigit(c)
char c;
{
        return('0' <= c && c <= '9');
}
```

```
/* Some functions for the example programs
   from chapter four
*/



/* functions handling DEQUES */
/* ========================= */

init_deque(d) struct deque *d;
{ d->left=DQL/2 + 1; d->right=DQL/2;}

empty_deque(d) struct deque *d;
{ return(d->left > d->right); }


ins_r(d, el) struct deque *d; int el;
{  d->cont[++d->right]=el; }

ins_l(d, el) struct deque *d; int el;
{  d->cont[--d->left]=el; }

del_r(d) struct deque *d;
{ return( d->cont[d->right--] ); }

left(d) struct deque *d;
{ return( d->cont[d->left] ); }


right(d) struct deque *d;
{ return( d->cont[d->right] ); }

del_l(d) struct deque *d;
{ return( d->cont[d->left++] ); }
```

```
/* functions handling QUEUES */
/* ========================= */

init_queue(d) struct queue *d;
{ d->left=1; d->right=0;}


empty_queue(d) struct queue *d;
{ return(d->left > d->right); }


ins_q(d, el) struct queue *d; int el;
{  d->cont[++d->right]=el; }

left_q(d) struct queue *d;
{ return( d->cont[d->left] ); }

right_q(d) struct queue *d;
{ return( d->cont[d->right] ); }

del_q(d) struct queue *d;
{ return( d->cont[d->left++] ); }



/* DIVIDE&CONQUER PRIMITIVES */
/* ========================= */


twolog(n) int ;
{ int l = 0;
  while(n>1)
    { n /= 2; l++; }
  return(l);
}

size(p) int p;
{ return(p); }


solve_seq(p) int p;
{ return(p); }


combine(p1,p2) int p1,p2;
{ return(p1+p2); }

split(p,p1,p2) int p,*p1,*p2;
{ *p1=p/2; *p2 = p - *p1; }
```

```
/* rts.h

   some definitions to be included in rts.c and the c version of
   a DNP program
*/

#define EOF '\01'
#define EOM -1
#define SEP -2
#define DQL 100
#define QUL 100


int _f; /* used for forking */

typedef int inchan;
typedef int outchan;
struct channel {
        inchan i;
        outchan o;
};

struct deque { int left; int right; int cont[DQL];} ;

struct queue { int left; int right; int cont[QUL];} ;

struct queue _m; /* used for multiple channel assignment in survivals */
```

CHAPTER FOUR

THE COMPLEXITY OF DNP PROGRAMS

4.1. INTRODUCTION

This chapter presents a number of algorithms all programmed in the language DNP defined in the previous chapter. The algorithms (e.g. for matrix multiplication) are believed to be prototypical for dataflow computing and illustrate the criteria used for an evaluation of their efficiency. Section 4.2. is devoted to algorithms that have an essentially linear computation graph: sorting and matrix multiplication, and to an algorithm that uses a binary tree of processes to implement a general divide-and-conquer routine efficiently. The rest of the chapter is devoted to an appraisal of the expressive power of DNP. In section 4.3. we consider the limitations of the language. The main theorem is that not all (important) classes of computation graphs can be generated by DNP programs. In sections 4.4. to 4.6. a comparison is made with the standard complexity classes.

Dataflow algorithms can be classified according to the topology of their computation graphs. The graphs that can be generated by a certain DNP program coincide with the graphs produced by a context free graph grammar in the sense of graph grammar theory (see [77]). Therefore, algorithms with context free computation graphs can be expressed in DNP in the following way, using the *expand* mechanism:

- "grow" the graph according to the input data, and
- let the processes in the nodes of the graph perform their particular task.

Take, for example, systolic algorithms [57], most of which can be expressed in DNP even though their underlying computation model (systolic arrays) is synchronous instead of asynchronous. Systolic arrays are regularly structured networks of simple processing elements that rhythmically act on

streams of data passing through the network. To show that systolic algorithms can be expressed in DNP, consider the algorithm for a "systolic stack" as given by Kramer and van Leeuwen [55], originally due to Leiserson [59]. The design consists of a linear array of cells with an I/O connection to the environment left of the first cell (see figure 4.1.1.)

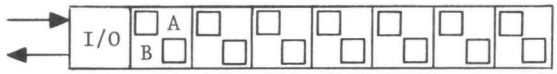


Figure 4.1.1. A systolic array.

Every cell has two registers, A and B, each of which can contain a number or a special *empty* token. The I/O cell is a passive cell, the registers of which can be set and inspected by the outside world. A *push* is represented by setting both the A and B register to a number, while a *pop* is represented by setting both the A and B register to empty. The systolic array is synchronized so that odd and even cells "beat" alternately. When it acts, a cell will inspect the registers of its left neighbour, which is inactive at that moment. When the left neighbour has numbers in both of its registers, one is copied into the active cell. When the left neighbour has two empty registers the active cell copies one into the neighbour. In this way pushes and pops ripple through the array without causing race hazards.

A dataflow program for a systolic stack neither has nor needs the global synchronization. Instead, the computation is controlled by the availability of tokens (the number itself for push, the empty token for pop) streaming through the array. A cell process has two inputs and two outputs (see figure 4.1.2.),

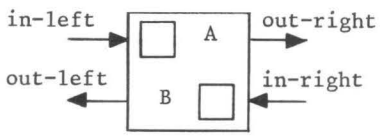


Figure 4.1.2. A cell-process.

and essentially performs the loop of figure 4.1.3. Note that a cell process only acts when it has a token (perhaps the empty token) on its in-left or in-right ports.

```
repeat
   read (in-left, A);
      if A = empty
         /* pop */
         then write (out-left, B);
              if B ≠ empty
              then write (out-right, empty);
                   read (in-right, B)
              fi
         /* push */
         else if B ≠ empty
              then write (out-right, B)
              fi;
              B:=A
      fi
forever
```

Figure 4.1.3.

Kramer and van Leeuwen prove that the systolic array can process push/pop commands in O(1) response times, as long as the number of elements in the stack remains less than the number of cells. This boundedness of the systolic algorithm can be overcome in DNP easily by having a "bumper" process at the right end of the array, which answers a pop command by sending an empty token to the left and a push command by expanding into a cell process that gets the pushed element and a bumper process (see figure 4.1.4.). Many other systolic algorithms can be translated to DNP in the same way, as long as their computation graph can be generated.

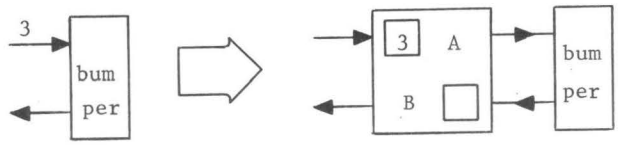


Figure 4.1.4. Expansion of a bumper process.

4.2. SOME DNP PROGRAMS AND THEIR COMPLEXITY

When analyzing a DNP program the following complexity measures can be used:

(i) *The number of processes.*

We can measure the total number of processes created during the whole computation, the maximal number of processes active at a certain moment, and the minimal number of processors needed to run a program. The last two measures are of interest if a processor can be reallocated when a process is no longer running on it or, if the creation of a process can be suspended until a processor becomes available.

(ii) *The number of channels and their size.*

Clearly the number of channels depends on the number of processes. The size of a channel at a certain moment is defined as the number of items written on the channel and not yet read from it. Hence the size of a channel depends on the timing of the algorithm.

(iii) *The number of time-steps necessary to execute the program.*

We will assume that all processes run in parallel and are equally fast, i.e., they perform the same DNP statement in the same number of time-steps. Such an execution could be characterized as "asynchronously synchronous".

4.2.1. A sorting program.

Parallel programs in general can be divided into (i) those where the input data is already in the parallel processes or memories, (ii) those where the input resides on a number of files and where the number of files depends on the size of the particular problem, and (iii) those where the input resides on a fixed number of files. Examples of programs in the first class are bitonic sorting [9] and a derivative of it that runs on a mesh-connected parallel computer [79]. A program in the second class is Kung's matrix multiplication on a hexagonal array of processors [57]. DNP programs fall in the third class and will therefore have a time complexity of at least O(n).

An interesting sorting algorithm in the third class is Todd's parallel merge sort [83] . This algorithm takes only log(n) processors to sort n numbers in about 2n+log(n) time-steps. In Todd's sort the passes of merge sort execute overlapped. Each pass resides on a separate processor, so one processor repeatedly combines single numbers into sorted runs of size two, the next processor repeatedly combines two runs of size two into one run of size four etc. (see figure 4.2.1.1.).

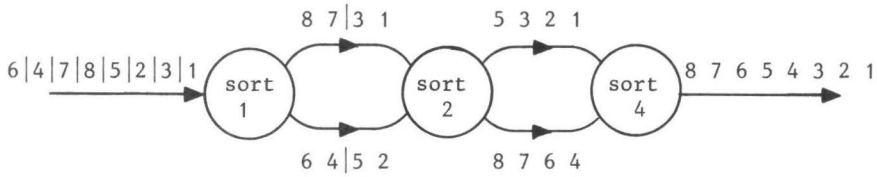


Figure 4.2.1.1. Todd's sort.

When the last number enters the first processor it will take O(log n) steps to get the first sorted number out of the sorting net.

The sorting algorithm we will present here is faster in the sense that immediately after the last number enters the net, the first number of the sorted sequence is output. This makes our sorting net easily adaptable to a priority queue that reacts on insert/delete commands in constant time. We will call it "pipeline sort".

The program starts as in figure 4.2.1.2., where *bottom* is a process doing nothing, i.e., sending an empty sequence over channel *r* to the process *sort*. The process sort will start reading elements of the unsorted sequence from channel *u*. The sorted sequence will eventually be written on channel *s*.

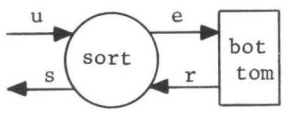


Figure 4.2.1.2. The initial sorting network.

Channel *e* never receives a token and is there for reasons of symmetry which will become clear below.

The sort process reads in and sorts elements in an internal datastructure, as long as it can do this in a constant time per element. Otherwise, it *expands* (see chapter three) into a subnet consisting of a new sort process (by means of a *creation*) and itself (by means of a *survival*). The newly created sort process takes over the reading and internal sorting of the unsorted sequence. In order to do this it has to gain control of the input channel u and the output channel s. The old sort process will, after the expansion, merge its internal sorted sub-sequence with a sorted sub-sequence coming from channel r. (For the first sort process the sorted sub-sequence from r will be empty.) The resulting (bigger) sorted sub-sequence will be sent to the newly created sort process over an intermediate channel *rr*. The channels necessary for the computation are drawn in figure 4.2.1.3.

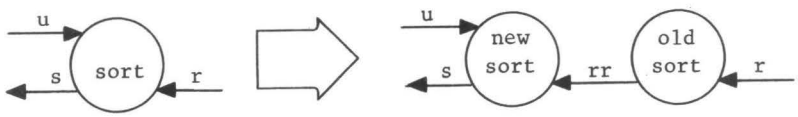


Figure 4.2.1.3. Necessary channels in sort expansion (but syntactically incorrect).

The expansion as pictured in figure 4.2.1.3. is however inexpressible in DNP, because processes must have the same channel configuration before and after expansion. For this reason the dummy channels e and *ee* are introduced (see figure 4.2.1.4.). We will come back to this phenomenon in section 4.3. where the limitations of DNP are discussed.

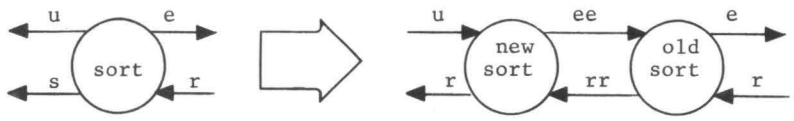


Figure 4.2.1.4. The syntactically correct expansion of sort.

Reading in elements and sorting them in constant time per element can be done in many ways. A possibility is to use a *deque* and to put elements that are greater or equal to the maximal element on one end and elements that are less or equal to the minimal element on the other end, and to stop when an element arrives that falls in between. This requires a flexible random access structure inside the sort process. We will see when analysing the program that this gives no great advantage as the average number of elements sorted internally in this way will be less than 5. A much simpler way is to allow a fixed number of elements to be internally sorted per process. An interesting number is *one*, because it will eliminate the need for internal sorting altogether. The pictures of figure 4.2.1.5. exemplify sorting the file 5,1,2,4 when the internal sorting is done using a deque.

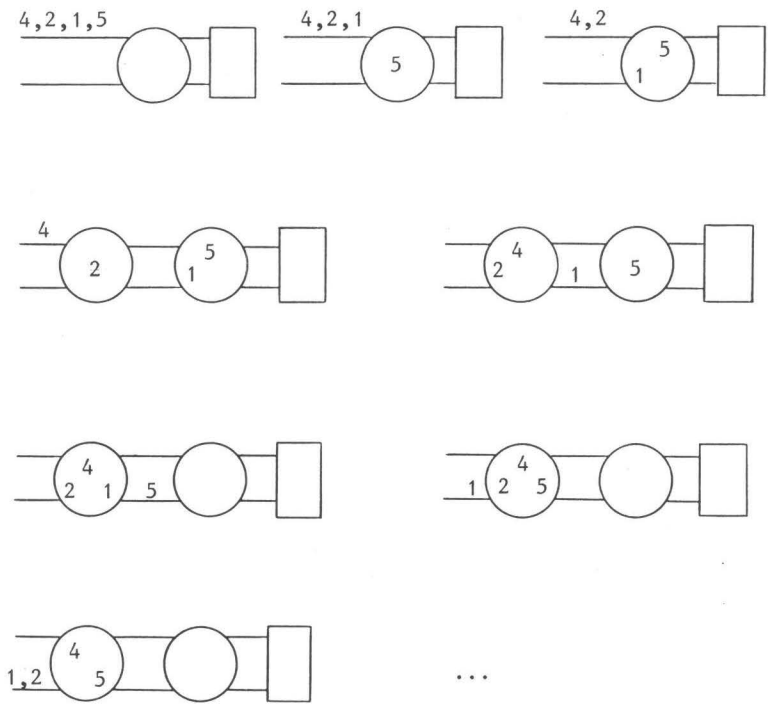


Figure 4.2.1.5. Pipeline sort in action.

Figure 4.2.1.6. shows the corresponding program and figure 4.2.1.7. shows the program where every process keeps only one internal number.

```
process sort(in u,r out s,e int f):
begin   int i;
        struct deque d; init_deque(&d); ins_l(&d,f);

        while(read_int(u,&i))
           { if(i <= left(&d)) ins_l(&d,i);
           else if (i >= right(&d)) ins_r(&d,i);
              else {
                   expand chan ee,rr
                     create sort(in u,rr out s,ee int i)
                     keep   sort(in ee,r out rr,e)
                   endexp;
                   break;
                   }
           }
        while(read_int(r,&i))
           {while(!empty_deque(&d) && (i >= left(&d)))
                write_int(s,del_l(&d));

            write_int(s,i);
           }
        while(!empty_deque(&d)) write_int(s,del_l(&d));
end


process bottom(in e out r):      begin   end

main m(in unsorted out sorted):
begin
        int i;
        if(read_int(unsorted,&i))
                { expand chan e,r
                     create bottom(in e out r)
                     create sort(in unsorted, r out sorted, e int i)
                  endexp;
                  }
end
```

Figure 4.2.1.6. Pipeline sort using a deque.

```
process sort(in u,r out s,e):
begin   int i,j;
        if(read_int(u,&i))
           {expand chan ee,rr
                   create sort(in u,rr out s,ee)
                   keep   sort(in ee,r out rr,e)
            endexp;

            while(read_int(r,&j))
             {if(j<i) write_int(s,j);
              else {write_int(s,i);
                    i=j;
                   }
             }
            write_int(s,i);
           }
        else while(read_int(r,&j)) write_int(s,j);
end


process bottom(in e out r):      begin   end



main m(in unsorted out sorted):
begin
        expand chan e,r
               create bottom(in e out r)
               create sort(in unsorted, r out sorted, e)
        endexp
end
```

Figure 4.2.1.7. Pipeline sort with only one internal element.

4.2.1.1. Analysis of pipeline sort.

The number of processes

The number of processes generated by pipeline sort with single numbers internally (figure 4.2.1.7.) is simply n+2: one bottom process, one process that detects end-of-file and one process for every number to be sorted. In the deque version (figure 4.2.1.6.) the number of processes depends on the order of the numbers on input. If the sequence is, e.g., already sorted, the number is two: one bottom process and one sort process. On the other hand, the number of processes is $\lceil n/2 \rceil+1$ when every (1+2i)-th number falls in between the preceeding two. In order to determine the average number of processes, we define a *semirun* as a sub-sequence of the unsorted sequence that can be sorted using one deque. Any sequence can be divided into a number of semiruns. Semiruns have a size of at least two.

Definition 4.2.1.1. Let I be a sequence of numbers $v_1,v_2,..$ $(1 \leq v_i \leq N)$. I models the unsorted sequence of numbers, v_1 is the first number to be read. A *sub-sequence* I[1..u], (1<u), is defined as the sequence $v_1,...,v_u$. The longest sub-sequence I[1..u] such that for all k from 1+1 to u

either $v_j \leq v_k$ for all j from 1 to k-1
or $v_j \geq v_k$ for all j from 1 to k-1

is called a *semirun*. A *prefix* is any initial sub-sequence of a *semirun*.

□

Thus a semirun is the longest sub-sequence such that each subsequent number is either less than or greater than all the previous numbers in the semirun.

We will determine the average length of the first semirun I[1..u] of a sequence I assuming that the numbers in the sequence are uniformly distributed over 1 to N, i.e., $P(v_i=n) = 1/N$ for all n from 1 to N. We write $P_N(k)$ for the probability that u=k by a given N, and $N_k(i,j)$ for the number of prefixes $v_1,...,v_k$ such that $\min(v_1,...,v_k) = i$ and $\max(v_1,...,v_k) = j$.

We then have

$$P_N(k) = \frac{\#\text{ sequences } v_1,\ldots,v_{k+1} \text{ such that } I[1..k] \text{ is a semirun}}{\text{total } \#\text{ sequences } v_1,\ldots,v_{k+1}}$$

$$= \frac{\sum\sum_{1\leq i\leq j\leq N} (N_k(i,j).(j-i-1))}{N^{k+1}} \qquad \text{(P1)}$$

where # stands for "the number of".

In order to determine $N_k(i,j)$ we observe the following:

$N_{k+1}(i,j)$ = # prefixes such that the last number (v_{k+1}) is equal to the old minimum or maximum (i.e., the bounds don't change)
+ # prefixes such that v_{k+1} is the new maximum
+ # prefixes such that v_{k+1} is the new minimum.

There are two cases: (1) i=j and (2) i<j. Using these cases and the above expression for $N_{k+1}(i,j)$ we can write down a recurrence relation for $N_{k+1}(i,j)$.

$$\left.\begin{aligned}
&(1)\ N_{k+1}(i,i) = N_k(i,i)\\
&(2)\ N_{k+1}(i,j) = 2N_k(i,j) \qquad (v_{k+1} = \min \text{ or } v_{k+1} = \max)\\
&\qquad + \sum_{l=i}^{j-1} N_k(i,l) \qquad (v_{k+1} = j)\\
&\qquad + \sum_{l=i+1}^{j} N_k(l,j) \qquad (v_{k+1} = i)\\
&\qquad = \sum_{l=i}^{j} (N_k(i,l) + N_k(l,j))
\end{aligned}\right\} \text{(N1)}$$

The basis of this recurrence relation is:

$$\left.\begin{aligned}
&(1)\ N_1(i,i) = 1\\
&(2)\ N_1(i,j) = 0
\end{aligned}\right\}$$

The above equations (N1) suggest that $N_{k+1}(i,j)$ depends on the values i and j independently, although it is intuitively clear that $N_{k+1}(i,j)$ depends on the number of values v_{k+1} (not) hitting the interval i to j, i.e., $N_{k+1}(i,j)$ depends on j-i only.

Lemma 4.2.1.1.1. $N_k(i,j)$ *depends on* j-i *only.*

Proof. By induction on k.

Base: $N_1(i,j)$ depends on j-i only.

Step: Suppose $N_k(i,j) = F_k(j-i)$ for some function F. Now check cases:

(1) $N_{k+1}(i,j) = F_k(j-i)$ if $(j-i) = 0$

(2) $N_{k+1}(i,j) = \sum_{l=i}^{j}(F_k(l-i) + F_k(j-l)) = 2\sum_{m=0}^{j-i} F_k(m)$

So in both cases $N_{k+1}(i,j)$ depends on (j-i) only.

□

If we define the function F_k by $F_k(j-i) = N_k(i,j)$, the equation (N1) transforms into:

$$F_{k+1}(n) = \begin{cases} 2\left(\sum_{j=1}^{n} F_k(j) + 1\right) & \text{if } 1 \leq n \leq N-1 \\ 1 & \text{if } n=0 \end{cases} \quad \text{(F1)}$$

Note that F_k is independent of N.

Lemma 4.2.1.1.2. $F_{k+1}(n)$, $n \neq 0$, *is a polynomial of degree* (k-1).

Proof. By induction on k.

Base: $F_1(n) = 0$

$F_2(n) = 2$

$F_3(n) = 2n+2$

Step: $F_k(n)$ is a polynomial of degree (k-2). Now observe that

$F_{k+1}(n+1) - F_{k+1}(n) = 2F_k(n+1)$.

It is well-known that if P(x) is a polynomial then

degree(P(x)) = d+1 $\Leftrightarrow$ degree(P(x+1)-P(x)) = d.

We therefore conclude that $F_{k+1}(n)$ is a polynomial of degree k-1.

□

We write $F_k(n) = \sum_{l=0}^{k-2} a_{kl}\, n^l$ for certain coefficients a_{kl} (k>1) and substitute it into (F1):

$$\begin{aligned} F_{k+1}(n) &= 2\left(\sum_{j=1}^{n} F_k(j) + 1\right) = 2\sum_{j=1}^{n}\left(\sum_{l=0}^{k-2} a_{kl}\, j^l\right) + 2 \\ &= 2\sum_{l=0}^{k-2} a_{kl} \sum_{j=1}^{n} j^l + 2 \end{aligned} \quad \text{(F2)}$$

From the theory of Bernoulli-polynomials and Bernoulli-numbers [52] we use the following facts:

(1) $B_n(x) = \sum_k \binom{n}{k} B_k \, x^{n-k}$ Bernoulli-polynomials

(2) $B_n = B_n(0) = B_n(1)$, $B_0 = 1$ Bernoulli-numbers

(3) $B_n(x+1) - B_n(x) = nx^{n-1}$

(4) $\sum_{p=1}^{n} p^l = \begin{cases} \frac{1}{l+1}(B_{l+1}(n+1) - B_{l+1}) & \text{if } l \geq 1 \\ n & \text{if } l = 0 \end{cases}$

Substituting property (4) into equation (F2) yields:

$$F_{k+1}(n) = 2 + 2\left(\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1}(B_{l+1}(n+1) - B_{l+1}) + a_{k0}\, n\right)$$

$$= 2 + 2a_{k0}\, n - 2\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1} + 2\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1}(n+1)$$

$$= 2 + 2a_{k0}\, n - 2\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1} + 2\sum_{l=1}^{k-2} a_{kl}\, n^l + 2\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1}(n)$$

Further manipulation shows:

$$\sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1}(n) = \sum_{l=1}^{k-2} \frac{a_{kl}}{l+1}\left(\sum_{j=0}^{l+1} \binom{l+1}{j} B_j \, n^{l+1-j}\right) \qquad \text{(by (1))}$$

$$= \sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} \sum_{p=0}^{l+1} \binom{l+1}{l+1-p} B_{l+1-p} \, n^p \qquad (p = l+1-j)$$

$$= \sum_{p=2}^{k-1} n^p \left(\sum_{l=p-1}^{k-2} \frac{a_{kl}}{l+1}\binom{l+1}{l+1-p} B_{l+1-p}\right) +$$

$$+ \sum_{l=1}^{k-2} \frac{a_{kl}}{l+1} B_{l+1} + \sum_{l=1}^{k-2} a_{kl} B_1 n$$

Therefore:

$$F_{k+1}(n) = 2 + 2n\left(a_{k0} + \sum_{l=1}^{k-2} a_{kl} B_1\right) + 2\sum_{l=1}^{k-2} a_{kl}\, n^l +$$

$$+ 2\sum_{l=2}^{k-1} n^l \left(\sum_{p=l-1}^{k-2} \frac{a_{kp}}{p+1}\binom{p+1}{p+1-l} B_{p+1-l}\right)$$

We conclude:

$$a_{k+1,0} = 2$$

$$a_{k+1,1} = 2\left(a_{k0} + \sum_{1=1}^{k-2} a_{k1}B_1\right) + 2a_{k1}$$

$$a_{k+1,1} = 2a_{k1} + 2\sum_{p=1-1}^{k-2} \frac{a_{kp}}{p+1}\binom{p+1}{p+1-1}B_{p+1-1} \qquad 2\leq 1\leq k-2$$

$$a_{k+1,k-1} = 2\,\frac{a_{k,k-2}}{k-1}\,.$$

And therefore:

$$a_{k,k-2} = \frac{2^{k-1}}{(k-2)!} \qquad \text{(because } B_0=1\text{)}$$

Now we substitute $F_k(n)$ into equation (P1):

$$P_N(k) = \frac{1}{N^{k+1}} \sum_{n=1}^{N-1} F_k(n)(N-n).(n-1)$$

$$= -\frac{N\sum_{n=1}^{N-1} F_k(n)}{N^{k+1}} - \frac{\sum_{n=1}^{N-1} F_k(n).n^2}{N^{k+1}} + \frac{(N+1)\sum_{n=1}^{N-1} F_k(n).n}{N^{k+1}} \qquad \text{(P2)}$$

$$= - A_{Nk} - B_{Nk} + C_{Nk}$$

Definition 4.2.1.1.2. $P(k) = \lim_{N\to\infty} P_N(k)$.

□

Theorem 4.2.1.1.3. $P(k) = \dfrac{2^k\,(k-1)}{(k+1)!}$ for $k\geq 2$.

Proof. By (P2) we have that $P_N(k) = - A_{Nk} - B_{Nk} + C_{Nk}$.

$A_{Nk} = O(1/N)$ because $F_k(n)$ is a polynomial of degree k-2.

We use the following notation: $Z(n,r) = \sum_{j=1}^{n} j^r = \frac{1}{r+1}\, n^{r+1} + O(n^r)$. Then

$$B_{Nk} = \frac{1}{N^{k+1}} \sum_{n=1}^{N-1} \left(\sum_{j=2}^{k} a_{k,j-2}\, n^j\right) = \frac{1}{N^{k+1}} \sum_{j=2}^{k} (a_{k,j-2}\, Z(N-1,j))$$

$$= \frac{1}{N^{k+1}}\, a_{k,k-2}\, Z(N-1,k) + O(1/N).$$

$$C_{Nk} = \frac{N+1}{N^{k+1}} \sum_{n=1}^{N-1} \left(\sum_{j=1}^{k-1} a_{k,j-1}\, n^j \right) = \frac{N+1}{N^{k+1}} \sum_{j=1}^{k-1} a_{k,j-1}\, Z(N-1,j)$$

$$= \frac{1}{N^k}\, a_{k,k-2}\, Z(N-1,k-1) + O(1/N).$$

$$P_N(k) = \frac{a_{k,k-2}(Z(N-1,k-1).N - Z(N-1,k))}{N^{k+1}} + O(1/N)$$

$$= a_{k,k-2}\left(\frac{1}{k} - \frac{1}{k+1}\right) + O(1/N) = \frac{a_{k,k-2}}{k(k+1)} + O(1/N)$$

$$= \frac{2^{k-1}\,(k-1)}{(k+1)!} + O(1/N).$$

Hence $P(k) = \dfrac{2^{k-1}\,(k-1)}{(k+1)!}$.

□

If we had assumed all numbers in the sequence to be unequal, i.e., $v_i \neq v_j$ $(i \neq j)$, the argument would have been much simpler. If there are p permutations of n unequal numbers such that they can be sorted in one deque, there are 2p such permutations of n+1 unequal numbers. As there are 2 permutations of 2 unequal numbers, there are 2^{k-1} such permutations of k unequal numbers and therefore

$$P(\text{first semirun has length} \geq k) = \frac{2^{k-1}}{k!}$$

or

$$P(\text{first semirun has length } k) = \frac{2^{k-1}}{k!} - \frac{2^k}{(k+1)!} = \frac{2^{k-1}\,(k-1)}{(k+1)!}$$

In order to determine the average length of the first semirun and its standard deviation we use a moment generating function [4]:

$$M(t) = \sum_{k=1}^{\infty} P(k)e^{kt} = \sum_{k=1}^{\infty} \frac{2^{k-1}\,(k-1)}{(k+1)!}\, e^{kt}$$

$$= e^t \sum_{k=1}^{\infty} \frac{(k-1)(2e^t)^{k-1}}{(k+1)!} = e^t \left(\sum_{k=1}^{\infty} \frac{(2e^t)^{k-1}}{k!} - 2\sum_{k=1}^{\infty} \frac{(2e^t)^{k-1}}{(k+1)!} \right)$$

$$= \frac{1}{2}(e^{2et} + 1) - \frac{1}{2e^t}\,(e^{2et} - 1).$$

The mean length is $M'(0) = \frac{1}{2}(e^2-1) \approx 3.19$

The standard deviation is $\sqrt{M''(0) - (M'(0))^2} \approx 1.17$

Determining the average length of the i-th semirun becomes horribly complicated because it depends on the previous semirun. The distribution of the first number of a semirun is not uniform and differs per semirun, though the distribution of the other numbers is again uniform. If we assume that the first number has no effect on the length of the semirun, because the second and third fall around it, the average length of the semirun will be 4.19, (1 for the first plus 3.19 for the uniformly distributed rest). So we conclude that the average length of any semirun will be less than 4.19. The above analysis shows that sorting with a flexible internal sorting structure such as a deque is not worthwhile.

The number of time-steps.

We will now analyse the number of time-steps used. First we consider pipeline sort with single numbers internally. We can distinguish three types of processes (see Figure 4.2.1.1.1.):

(1) one bottom process
(2) n internal sort processes
(3) one last process detecting end-of-file.

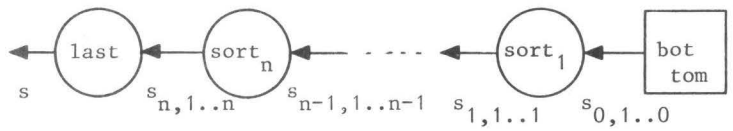


Figure 4.2.1.1.1.

From the program text in figure 4.2.1.7. it is clear that a sort-process performs at most 2 ordinary statements between reading, writing and expanding. We will therefore only take communication and expand statements into account. We will assume that writing, expansion, reading an already available number, and reading from a permanently empty channel will each take one time-step, and reading from a temporarily empty channel will be finished one time-step after

a number has been written on the channel by the producing process.
An internal $sort_i$ process goes through the following stages:

(1) create (c)
(2) read i-th number from unsorted sequence (ru_i)
(3) keep (k)
(4) merge i-th number into sorted sub-sequence $s_{i-1,1..i-1}$ and write out sorted sub-sequence $s_{i,1..i}$

The last sort process goes through the following stages:

(1) create (c)
(2) read from empty channel of unsorted sequence (ru_e)
(3) copy sorted sequence $s_{n,1..n}$ to outside world (writing number j to outside world: ws_j)

We will use the following additional notation:

reading number j from sub-sequence s_{i-1}: $rs_{i-1\ j}$
writing number j on sub-sequence s_i: ws_{ij}
reading from empty channel s_{i-1}: $rs_{i-1\ e}$

A typical execution is shown in figure 4.2.1.1.2. Note that the last sort process has to wait one time-step for the first number from the sorted sequence.

Proposition 4.2.1.1.6. *Pipeline sort with single numbers internally reads the unsorted sequence in* O(1) *time-steps per number and writes out the sorted sequence in* O(1) *time-steps per number immediately afterwards.*

Proof. $Sort_i$ is created at time-step 2i-2 and reads its number v_i at time-step 2i-1, so reading proceeds at O(1) time-steps per number. Writing the j-th number on the i-th sorted sub-sequence (ws_{ij}) is done at time-step 2(i+j) by $sort_i$; the same number is read by $sort_{i+1}$ at time-step 2(i+j)+1 ($i \neq n$). The last sort process is created at time-step 2n; it reads from s_n at time-step 2n+2 and receives the first number (rs_{n1}) at time-step 2n+3. The j-th number from s_n is read at time-step 2(n+j)+1 and written at time-step 2(n+j)+2.
□

time steps

processes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$sort_1$	c	ru_1	k	rs_{0e}	ws_{11}																		
$sort_2$			c	ru_2	k	rs_{11}	ws_{21}	rs_{1e}	ws_{22}														
$sort_3$					c	ru_3	k	rs_{21}	ws_{31}	rs_{22}	ws_{32}	rs_{2e}	ws_{33}										
$sort_4$							c	ru_4	k	rs_{31}	ws_{41}	rs_{32}	ws_{42}	rs_{33}	ws_{43}	rs_{3e}	ws_{44}						
$sort_5$									c	ru_5	k	rs_{41}	ws_{51}	rs_{42}	ws_{52}	rs_{43}	ws_{53}	rs_{44}	ws_{54}	rs_{4e}	ws_{55}		
last											c	ru_e	rs_{51}	rs_{51}	ws_1	rs_{52}	ws_2	rs_{53}	ws_3	rs_{54}	ws_4	rs_{55}	ws_5

Figure 4.2.1.1.2. A time diagram for n=5.

We will now analyse the timing of pipeline sort with deques internally. Again we assume the same timing for reading, writing and expanding, and neglect the rest of the statements because there are only a (small) constant number of them between reading, writing and expanding. An internal sort process reads at least two numbers from the unsorted sequence so the total number of reads and writes on (internal) sorted sub-sequences will be less than in the previous case. The important property is that when an internal sort process reads from the channel r containing the sorted sub-sequence either a number is immediately available or the channel is permanently empty. The last sort process reads zero or more numbers from the unsorted sequence. Time diagram 4.2.1.1.3. shows a worst case where 5 numbers are sorted by three sort-processes with deque-lengths 2,2 and 1 respectively.

time-steps

		0	1	2	3	4	5	6	7	8 . . .
P										
r	main	ru_1								
o	bottom		c							
c	$sort_1$		cu_1	ru_2	ru_3	k	rs_{0e}	ws_{11}	ws_{12}	
e	$sort_2$					cu_3	ru_4	ru_5	k	3*r, 4*w
s	last								cu_5	ru_e 5*r, 5*w
s										

Figure 4.2.1.1.3. A time diagram for sort using deques.

Proposition 4.2.1.1.7. *Pipeline sort with deques internally reads its input numbers in* O(1) *time-steps per number and outputs the sorted sequence in* O(1) *time-steps per number immediately afterwards.*

Proof. Process $sort_i$ is created at time-steps $c_i = \sum_{j=1}^{i-1}(dq_j+1)$ where dq_j is the length of the j-th deque. $Sort_i$ then reads dq_i numbers and performs a keep at time-step $k_i = c_i + dq_i + 1$, and then performs $(\sum_{j=1}^{i-1} dq_j + 1)$ reads and $\sum_{j=1}^{i} dq_j$ writes mixed through each other such that there is at least one write after

each read so time-step $(r_{i-1\ j}) \geq (k_i+2j-1)$ and time-step $(w_{ij}) \leq (k_i+2j)$. If $sort_i$ writes to another internal sort process $sort_{i+1}$, which is created at $c_{i+1} = k_i$, we have that $dq_{i+1} \geq 2$ and therefore $k_{i+1} \geq (k_i+3)$, which means that the numbers from the sorted sub-sequence are immediately available when needed. Now if $sort_i$ writes to the last process, last sort performs at least one read from the unsorted sequence and then starts merging after time-step (k_i+2) so the last sort process may have to wait one time-step for its first number after which the last sort process and $sort_i$ are synchronized.

□

The channel-sizes.

We shall analyze the size of the channels carrying the sorted sub-sequences in the case of pipeline sort with deques. The channel-size is defined as the number of items written but not yet read at a certain moment. These quantities are, in this case, stochastic.

Proposition 4.2.1.1.8. *In the worst case the* i*-th channel carrying sorted sub-sequences* s_i *assumes the size* $\sum_{j=1}^{i} dq_j$.

Proof. For this to happen $sort_{i+1}$ must read and sort its internal numbers into $deque_{i+1}$ in at least as many time-steps as it takes $sort_i$ to put out its sorted sub-sequence. As $sort_{i+1}$ takes one time-step for reading and sorting one number, and $sort_i$ must perform $\sum_{j=1}^{i} dq_j$ writes, the worst case occurs when

$$dq_{i+1} \geq 2\sum_{j=1}^{i-1} dq_j + dq_i$$

□

Again we must conclude that using a flexible structure for internal sorting and thereby introducing unpredictable behaviour of the program is not really worthwhile.

4.2.2. Matrix multiplication.

In this section we will consider an algorithm for matrix multiplication, restricted to square matrices. The matrix multiplication algorithm designed by Kung [57] has the form of a hexagonal grid (see figure 4.2.2.1.).

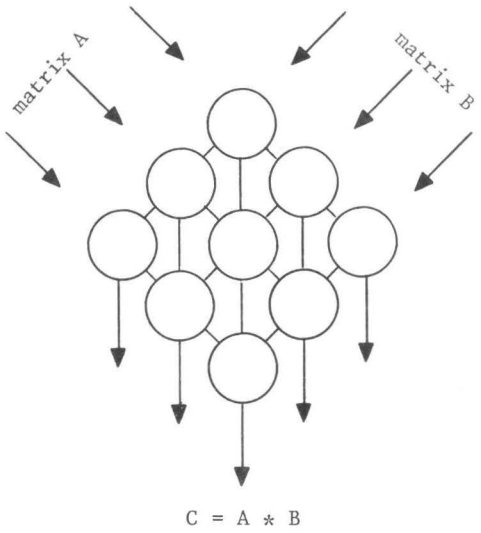


Figure 4.2.2.1. Kung's systolic matrix multiplication.

This algorithm needs O(n) connections to the outside world to multiply n by n matrices, which makes it unsuitable for VLSI implementation. The algorithm presented here needs only a constant number of connections to the outside world: two input channels and one output channel. Both input channels contain an input matrix. One matrix is in row format and the other is in column format. A matrix in row (column) format is a sequence of rows (columns) closed by an end of matrix mark (EOM or *). A row (column) is a sequence of numbers preceded by a begin of row (column) mark (SEP or $). The program will deliver the output in row format.

If there are n rows and columns the dataflow net will expand into two linear branches connected by a "central process" (see figure 4.2.2.2.)

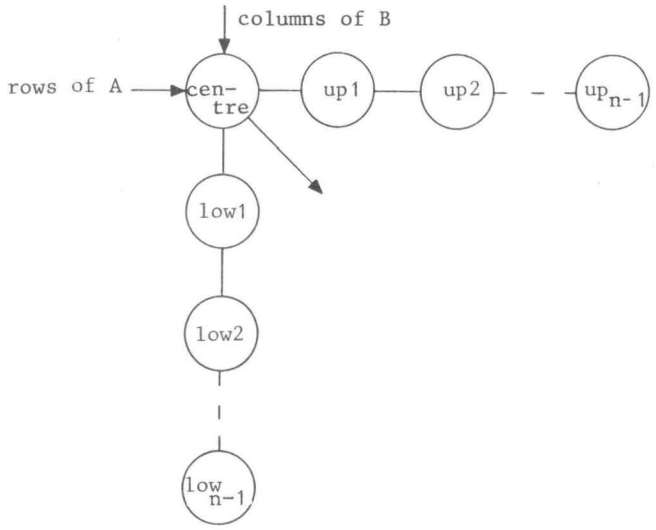


Figure 4.2.2.2. Basic form of a matrix multiplier.

Every process will compute a *diagonal* of the product matrix by traditional means. The centre-process will compute the central diagonal. The i-th up-process will compute the i-th upper diagonal, the i-th low-process will compute the i-th lower diagonal. In order to compute these diagonals centre needs all rows of A and columns of B, upi needs the first to (n-i)-th row of A and the (i+1)-th to n-th column of B, lowi needs the (i+1)-th to n-th row of A and the first to (n-i)-th column of B.

An important part of the program is concerned with getting the rows and columns where they are needed. Figure 4.2.2.3. shows how this is done for n=3. The general case is analoguous. The □ processes are duplicators. A row duplicator (dr) sends the first row it receives to the right and the rest of the rows both to the right and down. A column duplicator (dc) sends the first column it receives down and the rest of the columns both down and to the right.

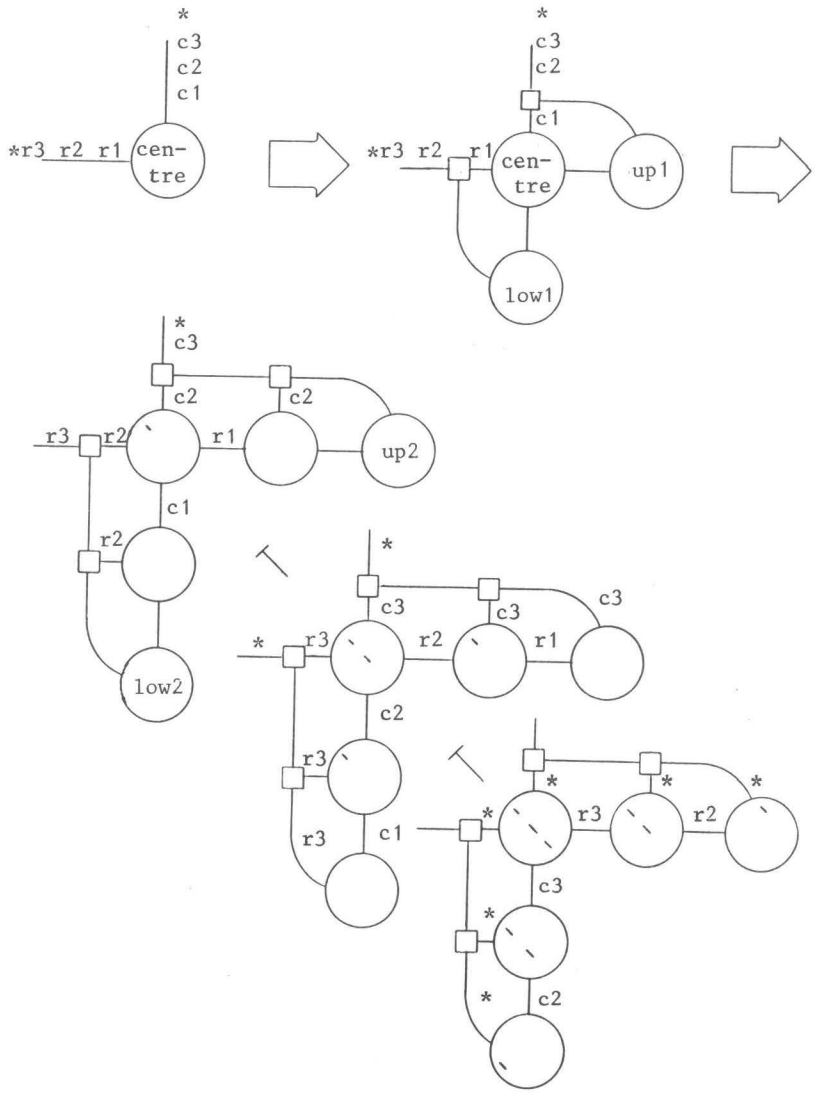


Figure 4.2.2.3. Distribution of rows and columns
in the 3*3 matrix multiplication.

After the diagonals have been computed, they are sent back to centre such that this process can format the product matrix in row order. The process of sending back and reformatting diagonals into row format differs slightly for the up and low processes. An up process first sends its i-th diagonal element to the left and then copies the row part from the right to the left. A low process will also form row parts. Figure 4.2.2.4. shows how the reformatting takes place when n = 3 and product-matrix-element(i,j) = 10*i+j.

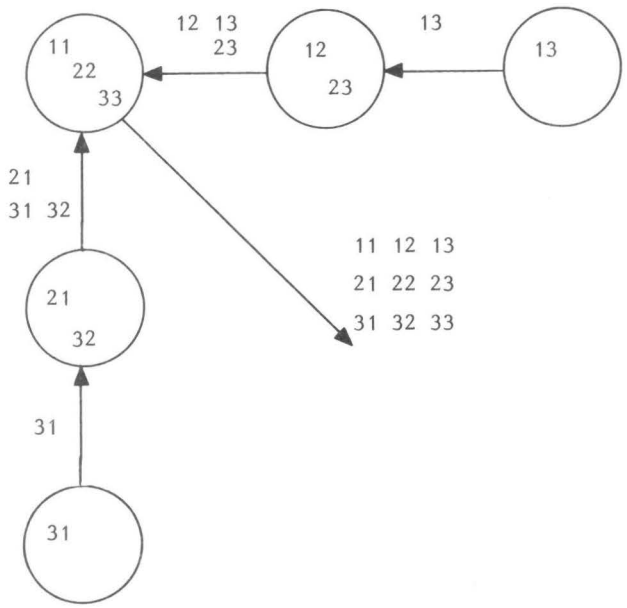


Figure 4.2.2.4. Reformatting the product matrix.

The above pictures are incomplete in that not all the required channels have been drawn. Figure 4.2.2.5. sketches an expansion into horizontal direction with all channels involved, where the process up0 expands into the encircled subgraph.

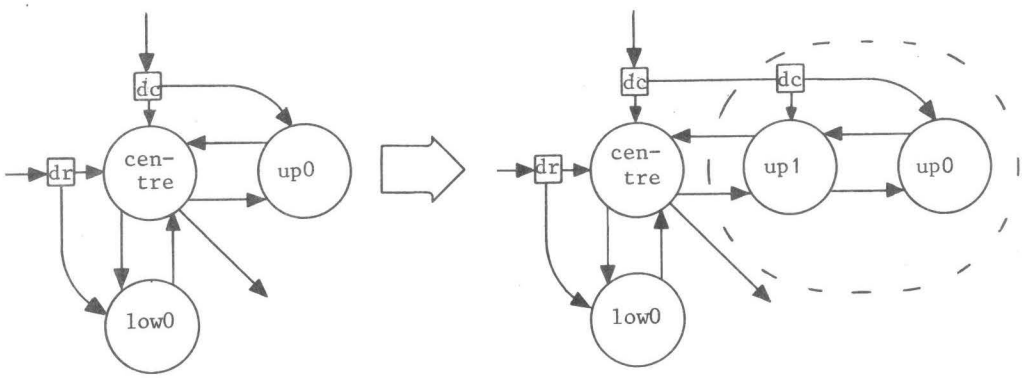


Figure 4.2.2.5. Complete expansion of an up0 process.

Figure 4.2.2.6. gives the complete program. The program starts as in the left-hand side of figure 4.2.2.5. The up0 and low0 processes are introduced to perform the expansion and do not take part in the computation of the diagonals.

```
process centre (in Mr,Mc,uptriangle,downtriangle
                out outrows,outcols,result):
begin
        int a,b,x;
        struct queue q;
        x = 0; init_queue(&q);


        write_item(outrows,SEP); write_item(outcols,SEP);

        /* create the middle diagonal of the product matrix */

        a = read_item(Mr); b = read_item(Mc);
        write_item(outrows,a); write_item(outcols,b);
        while (a!=EOM)
                {
                if (a==SEP)
                        { ins_q(&q,x); x = 0; }
                else x +=  a*b;
                write_item(outrows,a=read_item(Mr));
                write_item(outcols,b=read_item(Mc));
                } ins_q(&q,x);

        /* collect upper and lower triangles */
        a = read_item(uptriangle);
        if (a==EOM)    /* we had a 1*1 "matrix" */
                {
                write_item(result,SEP);
                write_item(result,del_q(&q));
                write_item(result,EOM);
                }

        else           /* form complete rows out of upper and lower triangles */

                do {write_item(result,SEP);
                    while ((b=read_item(downtriangle))!=SEP && b!=EOM)
                                write_item(result,b);
                    write_item(result,del_q(&q));
                    if (b != EOM)
                        while ((a=read_item(uptriangle))!=SEP && a != EOM)
                                    write_item(result,a);
                   } while (b != EOM);

        write_item(result,EOM);

end
```

```
process up0(in incols,inrows out res):
begin
        int a,b;
        a = read_item(incols);
        if (a==EOM)  write_item(res,EOM);

        else { b = read_item(inrows);
               if (a==SEP)
                   expand chan Mc1,Mc2,outr,upt
                        create dup(in incols out Mc1,Mc2)
                        create up0(in Mc2,outr out upt)
                        create up(in Mc1,inrows,upt out outr,res)
                   endexp
             }
end


process up(in incols,inrows,intriangle out outrows,outtriangle):
begin
        int a,b,x; struct queue q;
        x=0; init_queue(&q);

        write_item(outrows,SEP);

        /* form an upper diagonal */
        a = read_item(incols); b = read_item(inrows);
        while (a!=EOM)
                {
                write_item(outrows,b);
                if (a==SEP) { ins_q(&q,x); x=0; }
                else x += a*b;
                a = read_item(incols); b = read_item(inrows);
                }
        write_item(outrows,EOM);
        ins_q(&q,x);


        /* send up the upper triangle in row format */

        while ((a=read_item(intriangle)) != EOM)
        {
                if (a==SEP)
                        {
                        write_item(outtriangle,SEP);
                        write_item(outtriangle,del_q(&q));
                        }
                else write_item(outtriangle,a);
        }
        write_item(outtriangle,SEP);
        write_item(outtriangle,del_q(&q));
        write_item(outtriangle,EOM);
end
```

```
process low0(in inrows,incols out res):
begin
        int a,b;
        a = read_item(inrows);
        if (a==EOM) write_item(res,EOM);
        else { b = read_item(incols);
               if (a==SEP)
                     expand chan Mr1,Mr2,outc,downt
                        create dup(in inrows out Mr1,Mr2)
                        create low0(in Mr2,outc out downt)
                        create low(in Mr1,incols,downt out outc,res)
                     endexp
             }
end


process low(in inrows,incols,intriangles out outcols,outtriangle):
begin
        int a,b,x; struct queue q;
        init_queue(&q); x=0;

        /* create lower diagonal */
        write_item(outcols,SEP);
        a = read_item(inrows);
        b = read_item(incols);
        while (a != EOM)
                {
                write_item(outcols,b);
                if (a==SEP)
                        { ins_q(&q,x); x=0; }
                else x += a*b;

                a = read_item(inrows); b = read_item(incols);
                }
        write_item(outcols,EOM);
        ins_q(&q,x);

        /* send up the lower triangle in row format */
        write_item(outtriangle,SEP);

        do {
                a = read_item(intriangle);
                if (a==SEP || a==EOM) write_item(outtriangle,del_q(&q));
                write_item(outtriangle,a);
           } while (a!=EOM);
end
```

```
process dup(in a out b,c):
begin
        int i;

        /* copy first row or col to channel b */
        while ((i = read_item(a)) != EOM && i != SEP)
                write_item(b,i);


        /* copy the rest to channels b and c */
        if (i == EOM)
                { write_item(b,EOM); write_item(c,EOM); }
        else { while (i != EOM)
                { write_item(b,i); write_item(c,i);
                  i = read_item(a);
                }
               write_item(b,EOM); write_item(c,EOM);
             }
end




main madm(in Mr, Mc  out product):
        begin int a,b;
              a = read_item(Mr); b = read_item(Mc);
              if(a == EOM)
                write_item(product,EOM);
              else
                expand chan Mr1,Mr2,Mc1,Mc2,upt,downt,outr,outc
                        create dup(in Mr out Mr1,Mr2)
                        create dup(in Mc out Mc1,Mc2)
                        create up0(in Mc2,outr  out upt)
                        create low0(in Mr2,outc  out downt)
                        create centre(in Mr1,Mc1,upt,downt
                                      out outr,outc,product)
                endexp
        end
```

Figure 4.2.2.6. Matmul.

4.2.2.1. Analysis of Matmul.

The analysis of Matmul is straightforward as there is no randomness involved. If n=1 the net is as drawn in figure 4.2.2.1.1.

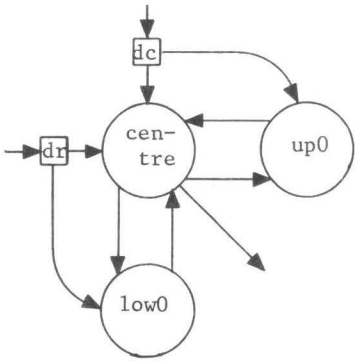


Figure 4.2.2.1.1. The net for n=1.

Proposition 4.2.2.1.1. *The number of processes ever created by Matmul is* 6n-1. *The maximum number of active processes at any moment is* 4n+1.

Proof. Figure 4.2.2.1.1. shows that initially there are 5 processes. If n>1, there will be (n-1) expansions of up0 and low0 processes (see figure 4.2.2.1.2.),

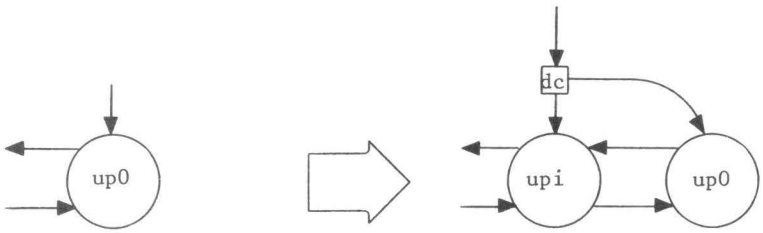


Figure 4.2.2.1.2. The i-th up0 process expanding.

causing 2*3*(n-1) process creations. The total number of processes ever created is therefore 6n-1. The maximal number of processes active at the same time is reached when all expansions have been performed. From that moment on the net consists of:

1 centre process 2n dup processes n-1 up processes n-1 low processes 1 up0 process 1 low0 process	which adds up to 4n+1 processes

□

An up or low process needs at most n storage locations for keeping its diagonal.

We will now analyse the number of time-steps needed to execute Matmul. Again we will only count reads, writes and expansions as the number of other statements between them is only a small constant.

Proposition 4.2.2.1.2. *Matmul takes* $O(n^2)$ *time-steps to read, multiply and write.*

Proof. The program starts executing as in figure 4.2.2.1.3., where rr, rc, w, e, and c stand for reading a row element, reading a column element, writing, expansion, and creation respectively. The moment of expansion of an up0(low0) process depends only on the input it receives from a dup process dr(dc).

	0	1	2	3	4	5 . . .
main	rr	rc	e			
centre			c			
dr1			c	rr	w	rr . . .
dc1			c	rc	w	rc . . .
up0			c			
low0			c			

Figure 4.2.2.1.3. First steps of Matmul.

The analysis of the timing of the expansion into the final net is therefore independent of the behaviour of centre, up and low processes. The timing of low0 processes is the same as that of up0 processes.

The first column is only sent to centre by dc1. Process up01 will receive its first item ($ or *) after O(n) time-steps. If n>1 up01 will expand and dc1 will send the columns to dc2 at the speed of one item per three time-steps. Except for the first column, which is written at the speed of one item per two time-steps, all other items are written at the speed of one item per three time-steps. The net will therefore be expanded after $O(n^2)$ time-steps.

The row items are sent to up1 through to up(n-1) by centre which will send a row element at the speed of one element per four time-steps after some initial waiting for input. Process up1 will get its row elements one per four time-steps and its column elements one per three time-steps so the pace of the whole net is determined by the slowest process: centre. Every process (centre, upi and lowi) performs an assignment

```
x += a*b
```

which implies that the diagonals are calculated in $O(n^2)$ time-steps.

The speed of the collecting phase is again dictated by the centre process, which reads and writes an item every two time-steps. We can conclude that the whole computation takes $O(n^2)$ time-steps.

□

The size of the channels carrying the rows from centre (upi) to up1 (upi+1) is at least n because the corresponding column will arrive at up1 (upi+1) only when centre (upi) has processed a complete row. All other channel sizes can be limited to one.

4.2.3. Divide-and-conquer algorithms.

In this section we will discuss an efficient implementation of *divide-and-conquer* algorithms on a tree of processors. The divide-and-conquer paradigm can be expressed as in figure 4.2.3.1.

```
proc div&co (problem p) answer r:
  begin if simple(p)
          then r = solve-simple(p)
          else problem p1, p2;
               split(p, p1, p2);
               r = combine(div&co(p1), div&co(p2))
        fi
  end.
```

Figure 4.2.3.1. The divide and conquer algorithm.

An interesting subclass of divide-and-conquer algorithms is the class of *recursive doubling* algorithms [78] where the divide-phase is not needed because the problem presents itself in an already divided form. The classical example is the calculation of $a_1+\ldots+a_n$. Figure 4.2.3.2. shows a computation graph (for n=8) that computes all partial sums $y_j = \sum_{i=1}^{j} a_i$ (j=1,..,8).

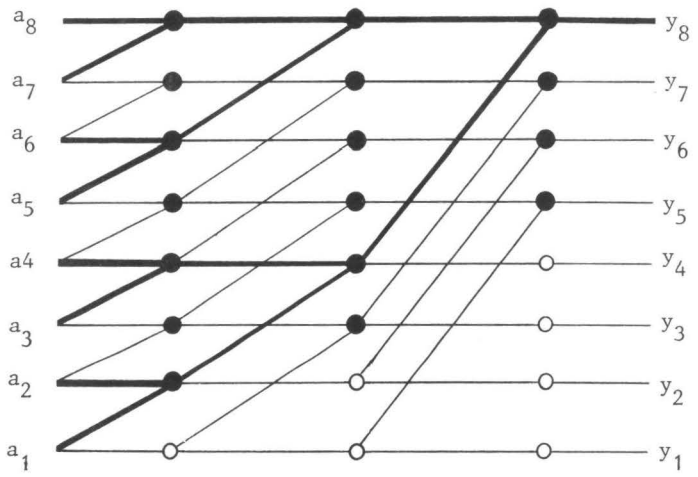


Figure 4.2.3.2. Summation of eight numbers.
Dark circles represent additions.
Open circles represent copy operations.

Stone [78] shows that the *inverse perfect shuffle* (see figure 4.2.3.3.) exactly provides the connections needed to evaluate recurrence relations of the form

$$y_0 = 0$$
$$y_j = y_{j-1} \underline{o}\ a_j \qquad (j \geq 1)$$

as long as $\underline{o}$ is an associative operator.

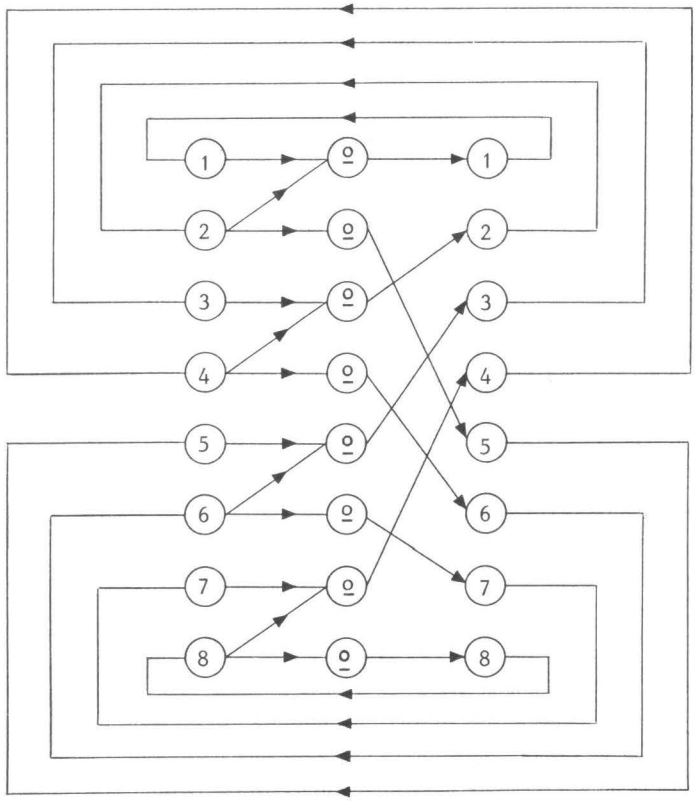


Figure 4.2.3.3. The inverse perfect shuffle of size 8.

If, in the example of addition, only the total sum $\sum_{i=1}^{8} a_i$ is needed, the calculation can be done using the much simpler interconnection structure of a tree of processes which we discuss here. Notice that in figure 4.2.3.2. this tree, indicated by thick lines, is a sub-structure of the complete computation graph.

Peters [68] discusses the implementation of divide-and-conquer on a binary tree machine. Communication between the processes is modelled as in CSP, which is equivalent to allowing channels of size zero in DNP. The calls *div&co(p1)* and *div&co(p2)* are executed in parallel on the two son processors of the processor running *div&co(p)*. Let the size of problem p be characterized by an integer n. For the time being we will assume n to be a power of two. The size of subproblems p1 and p2 is assumed to be n/2. A function g(n) denotes the time to execute the split and combine steps plus the time for parameter passing. The time s(n) required to execute the sequential version of divide-and-conquer is defined by the recurrence relation

$$s(1) = c$$
$$s(n) = 2s(n/2) + g(n)$$

while the time t(n) required to execute the parallel version is defined as

$$t(1) = c$$
$$t(n) = t(n/2) + g(n).$$

Assuming g(n) is a simple polynomial in n of the form αn^p, the recurrence relations have the solutions [68]:

$$\left.\begin{aligned} s(n) &= \frac{2^{p-1}}{2^{p-1}-1}\alpha n^p + c_1 n \\ t(n) &= \frac{2^p}{2^p-1}\alpha n^p + c_2 \end{aligned}\right\} \text{ for } p \neq 0,\ p \neq 1$$

$$\left.\begin{aligned} s(n) &= \alpha n \log n + c_1 n \\ t(n) &= 2\alpha n + c_2 \end{aligned}\right\} \text{ for } p = 1$$

$$\left.\begin{aligned} s(n) &= -\alpha + c_1 n \\ t(n) &= \alpha \log n + c_2 \end{aligned}\right\} \text{ for } p = 0$$

where c_1 and c_2 are constants.

From these solutions it follows that it is only worthwhile, in terms of execution time, to apply tree machines when $p \leq 1$. The most interesting case is the case p=0, where the run-time reduces from O(n) to O(log n).

The algorithm uses its processors rather inefficiently: only one level of the tree is active while the other processes wait for subproblems to be solved. We will discuss two improvements of the algorithm that do not change the order of t(n) but reduce the number of processes from 2n-1 to $\frac{n}{\log n}$.

The first improvement is to keep processes busy after they have submitted subproblems to their children. For that purpose a *special root process* is placed above the tree of divide-and-conquer processes (see figure 4.2.3.4.). The root process is special in that it does not submit both subproblems down the tree, but keeps one to itself to divide-and-conquer recursively. In the following step the subproblem it kept is split into two sub-subproblems, one of which it will send down and one of which it will keep again, etc. It should be observed that this technique is analoguous to tail-recursion removal.

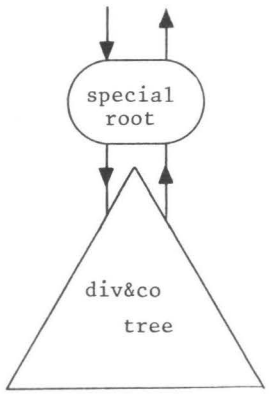


Figure 4.2.3.4. The divide-and-conquer net.

Figure 4.2.3.5. shows how a problem of size 8 is step-wise divided over eight processes, where pi..j denotes a problem of size j-i+1 and pi denotes a problem of size one. The tree grows with the size of the problem. Initially it will be as in figure 4.2.3.6.

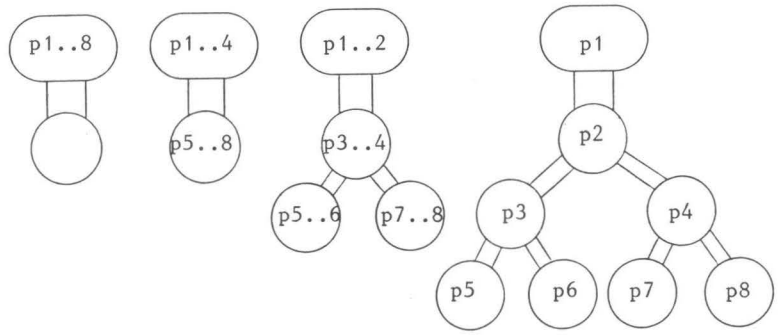


Figure 4.2.3.5. Divide-and-conquer in action.

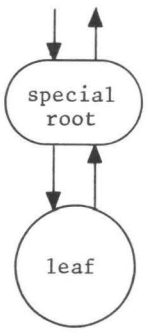


Figure 4.2.3.6. Initial state of the net.

A leaf process will, upon receipt of a problem, check whether the problem is simple or not. If it is simple the leaf process will solve the problem and send the result upwards, otherwise it will expand as in figure 4.2.3.7.

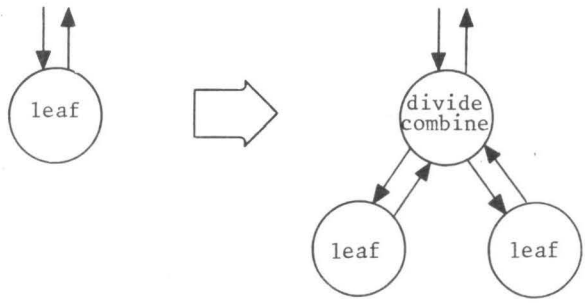


Figure 4.2.3.7. Expansion of a leaf process.

The divide-combine process in figure 4.2.3.7. will split the problem, send it down, wait for more problems to be split and sent down, solve a simple problem, send up the simple result, and combine and send up the results it gets from below. Figure 4.2.3.5. also outlines the timing of the algorithm. At time-step 2 to 4 the problem is split into simple problems. At time-step 5 all simple problems are being solved. At time-steps 6 to 8 the results are sent up and combined and at time step 9 the result is output.

Proposition 4.2.3.1. *The first improvement of the divide-and-conquer algorithm causes the algorithm to use only* n *processes while keeping the time complexity of the original algorithm.*

Proof. In the original version of the algorithm only leaf processes solve the simple problems. A perfect binary tree with n ($=2^k$) leafs contains 2n-1 nodes. In the first improvement of the algorithm every process solves a simple problem, so only n processes are needed. The time complexity of the algorithm does not change because the same actions are performed at the same time but by different processes.

□

Up till now we have assumed an ideal situation: the size of the original problem is a power of two. What happens if this is not so? A problem of size n is split in one of size $p_1 = \lfloor n/2 \rfloor$ and one of size $p_2 = \lceil n/2 \rceil$. Clearly $|p_1 - p_2| \leq 1$. Every division step will yield twice as many subproblems with at most two different sizes s_1 and s_2 such that $|s_1 - s_2| = 1$. After $\lfloor \log n \rfloor$ division steps all subproblem are of size one or two. Dividing the remaining problems of size two causes the tree of processes to be unbalanced and complicates the logic of the processes. The remaining problems of size two will therefore not be split anymore but solved sequentially.

The idea of not splitting the subproblems until their size is one can be exploited further. Every next division step yields just as many new processes as already present. It is therefore not worthwhile to keep splitting until the problem size is one or two. Consider the case where splitting and combining takes a constant time. The time complexity of the overall algorithm is then $O(\log n)$. In order to preserve this time complexity, problems will be split until their size is about log n and then solved sequentially. This can be implemented in combination with a method to keep the tree perfect: a process will be parameterized in a fashion that indicates how many problems it is going to have to split. This number is calculated by the root process and is spread and decreased through the tree. The root process will split $\lfloor \log n - \log\log n \rfloor$ problems and a divide-combine process on level i will split $\lfloor \log n - \log\log n \rfloor - i$ problems. This we call the second improvement to the divide-and-conquer algorithm.

Proposition 4.2.3.2. *The second improvement to the divide-and-conquer algorithm causes the algorithm to use* $O(\frac{n}{\log n})$ *processes. If the split and combine operations take constant time, the time complexity of the overall algorithm is* $O(\log n)$.

Proof. If we split the subproblems until their size is one or two we create a tree of processes of depth $\lfloor \log n \rfloor$ containing $2^{\lfloor \log n \rfloor}$ processes. Every lowest level of the tree of processes we save gains us half of the processes in use. Splitting the subproblems until their size is $\lfloor \log n \rfloor$ will therefore take $O(n - \frac{n}{2} - \ldots - \frac{n}{\log n}) = O(\frac{n}{\log n})$ processes (see figure 4.2.3.6.).

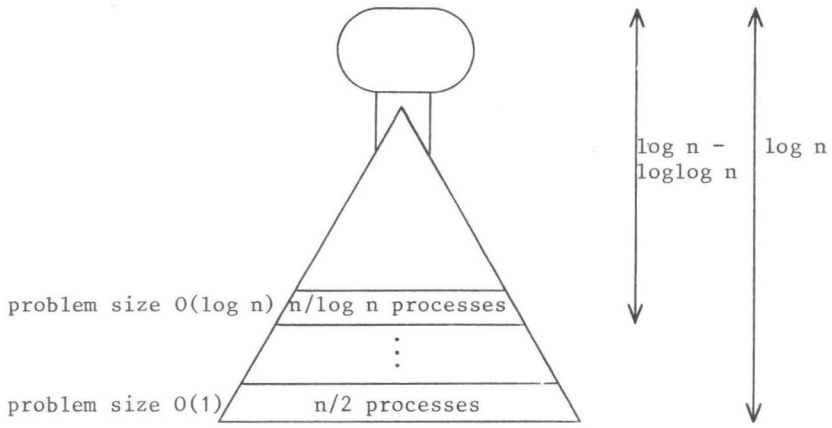


Figure 4.2.3.6. Saving more processes.

The time complexity of the algorithm now changes. First there will be O(log n - loglog n) timesteps to divide the problem into subproblems of size O(log n). Then these subproblems will be solved sequentially which takes s(log n) timesteps. As we have assumed that splitting and combining takes constant time, i.e., g(n) = α, we conclude that s(log n) = O(log n). Then the results are sent up which takes again O(log n - loglog n) time-steps. The overall time complexity of the algorithm will therefore remain O(log n).

□

As the sequential divide-and-conquer algorithm has time complexity O(n), the $\frac{n}{\log n}$ processes versus O(log n) timesteps is optimal, i.e., cannot be lowered without incurring a greater compute time.

Figure 4.2.3.7. shows a general divide-and-conquer program with both improvements incorporated. In this program problems and results are represented by integers. The primitive functions *size*, *split*, *solve-seq* and *combine* are assumed to be predefined. In our case they are part of the run-time system.

```
process root (in prb, subress out res, subprbs):
begin int i,c,n;
      int p,p1,p2;
      int s1,s2;
      read_int(prb,&p);
      n = size(p);
      if (n == 1) write_int(res,solve_seq(p));
      else
       { n /= twolog(n);
         c = n;
         write_int(subprbs,c/2);
         while((c /= 2) > 0)
          { split(p,&p1,&p2);
            write_int(subprbs,p2);
            p=p1;
          }
         s1=solve_seq(p);
         c = n;
         while((c /= 2) > 0)
           { read_int(subress,&s2);
             s1=combine(s1,s2);
           }
         write_int(res,s1);
       }
end
```

```
process leaf (in prbs out ress):
begin int p,c;
      if(read_int(prbs,&c))
      { if(c == 1)
         { read_int(prbs,&p);
           write_int(ress,solve_seq(p));
         }
        else
           expand chan subprbsleft, subprbsright,
                       subressleft, subressright
               create leaf(in subprbsleft out subressleft)
               create leaf(in subprbsright out subressright)
               create divco(in prbs, subressleft, subressright
                            out ress,subprbsleft, subprbsright
                            int c)
           endexp
      }
end
```

```
process divco (in prbs, subressleft, subressright
            out ress, subprbsleft, subprbsright
            int c):
begin   int i,p;
        int p1,p2,s,s1,s2;
        i = c;
        write_int(subprbsleft,c/2); write_int(subprbsright,c/2);

        while( (c /= 2) > 0)
                {
                read_int(prbs,&p);
                split(p,&p1,&p2);
                write_int(subprbsleft,p1);
                write_int(subprbsright,p2);
                }
        read_int(prbs,&p);
        write_int(ress,solve_seq(p));
        c = i;
        while( (c /= 2) > 0)
                {
                read_int(subressleft,&s1);
                read_int(subressright,&s2);
                s=combine(s1,s2);
                write_int(ress,s);
                }
end



main divconq (in prb out res):
begin   expand  chan subprbs, subress
                create root(in prb, subress
                            out res, subprbs)
                create leaf(in subprbs out subress)
        endexp
end
```

Figure 4.2.3.7. Divconq.

4.3. LIMITATIONS OF DNP

The following limitations of DNP are apparent:

(1) a process cannot change its channel configuration,
(2) there is no inverse of expansion: contraction,
(3) it is impossible to create all computation graphs.

We discuss various aspects of these limitations in the following subsections.

4.3.1. Changing the channel configuration.

The wish to change the channel configuration of a process presents itself naturally when programming in DNP. An extension is to allow a surviving process to *close* one or more channels. Consider as an example the pipeline sort algorithm of section 4.2.1. The *sort* processes have two input channels *u* and *r* and two output channels *s* and *e*. The *e* channel is connected to the *u* channel of the predecessor so that the rules for channel usage in an expansion are satisfied. What we like to express is that a *sort* process needs two input channels before expansion but needs only one input channel afterwards (see figure 4.3.1.1.).

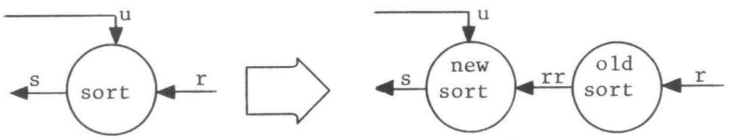


Figure 4.3.1.1. Changed channel configuration.

This can be expressed using a new keyword *close* as in figure 4.3.1.2.

```
process sort(in u, r out s):
  begin .
        expand chan rr
               create sort(in u, rr out s)
               keep   sort(in close, r out rr)
        endexp
        :
  end
```

Figure 4.3.1.2. An expand statement that closes a channel.

The problem with changing the channel configuration of a process is that the static check for correct channel usage in a (next) expansion does not work anymore. In the simple case of a closed channel in a surviving process we can consider the closed channel as a channel connected to a dummy process. In more complex cases of adding channels or creating a process with closed channels, the simple and elegant properties of the expand statement are lost. For this reason we have decided not to extend the language in this direction.

4.3.2. Contraction.

One can think of various types of contraction:

(i) Any subgraph can contract into a node, i.e., all nodes and all channels connecting these nodes together are replaced by one node connected to the rest of the graph by the channels that connected the old subgraph to the rest of the graph. An example of the use of such a contraction is the following. Suppose we have a problem P that can be solved recursively:

```
P(n,m) = H(P(n-1,m), P(n,m-1))
P(n,1) = ...
P(1,m) = ...
```

One could write down a divide-and-conquer style program for this problem as sketched in figure 4.3.2.1.

```
process solve-P(out r int n,m)
   begin.
        expand chan r1, r2
               create H(in r1, r2 out r)
               create solve-P(out r1 int n-1, m)
               create solve-P(out r2 int n, m-1)
        endexp
   end
```

Figure 4.3.2.1.

This is, however, rather inefficient because the subproblem P(n-1, m-1) is going to be solved twice (see figure 4.3.2.2.).

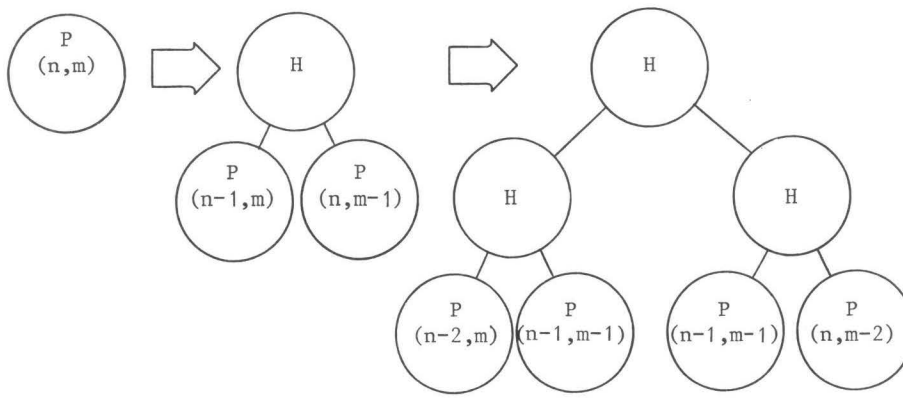


Figure 4.3.2.2. Inefficient divide-and-conquer solution for P.

Clearly one would like to "contract" the two P(n-1,m-1) processes. There are, however, serious problems with this type of contraction: (a) the locality principle is violated, (b) a process can wind up with an arbitrary number of channels, and (c) it is unclear how and where to define the process that will run in the newly created node.

(ii) One can allow a subgraph that was created in one expansion to contract back into the process from which it originated. This form of expansion is know as *parallel procedure calling* and is studied by Misra and Chandi [63]. In this model there is no place for surviving processes, because the state of a just contracted process is then ambiguous. So parallel procedure calling is an alternative to expansion rather then an extension.

(iii) A node can be *killed* if it does not execute further output instructions, or if all the processes it writes to are killed. This is in fact an implementation consideration and not a language feature. It can be compared to garbage collection in conventional languages.

4.3.3. It is impossible to create all computation graphs in DNP.

In this section we will reformulate the graph generating capabilities of DNP in graph grammar terminology and prove that there are important classes of graphs that cannot be generated. Similar work has been done for other types of grammars ([75],[33]). Our model of graph expansion turns out to be equivalent to Slisenko's version of context-free graph grammars [77].

Definition 4.3.3.1. A *star graph* is a pair <K,B> where

K is a graph, and

B a finite set of edges different from the edges of K, and every edge in B is connected to one node of K, though each node of K can be connected to zero or more edges of B. B can be empty.

K is called *the kernel* and B is called *the boundary*.

□

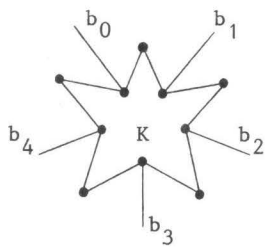


Figure 4.3.3.1. A star graph.

Figure 4.3.3.1. shows a star graph, where K consists of ten nodes and ten edges connecting these nodes and B consists of five edges b_0 to b_4 connecting K to the outside world. A *simple star* is a star graph where K consists of one node without edges.

Definition 4.3.3.2. A *context-free graph grammar (GFGG)* is a four-tuple $G = \langle N,T,P,S \rangle$ where

N and T are two disjoint finite alphabets for labelling *non-terminal* and *terminal nodes* respectively,
P is a finite set of *production rules*,
S is an element of N, the *starting symbol*.

Production rules are pairs (S_L,S_R), with $S_L = \langle K_L,B_L \rangle$ and $S_R = \langle K_R,B_R \rangle$ star graphs such that $|B_L| = |B_R|$ and S_L is a simple star. K_L is labelled with a non-terminal. All nodes in K_R are labelled too, with either terminal labels or non-terminal labels. Two production rules with the same K_L must have the same B_L. Productions are denoted as:

$$S_L \rightarrow S_R.$$

□

Figure 4.3.3.2. is an example of a production rule.

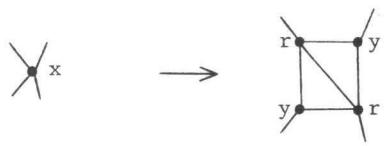


Figure 4.3.3.2. A production rule.

A context-free graph grammar G will be used to generate a class of labelled graphs through the process of "derivation". A *derivation step* according to a production rule $\langle X,B_L \rangle \rightarrow \langle K,B_R \rangle$ consists of replacing a node labelled X in a graph W by a subgraph K such that the boundary edges remain unchanged with respect to the nodes connected to the node labelled X in W. Clearly the subgraph K can be "glued" into W in many different ways. Figure 4.3.3.3. shows one possible derivation step.

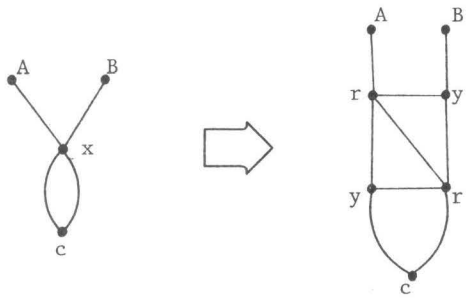


Figure 4.3.3.3. A derivation step using the production rule of figure 4.3.3.2.

Definition 4.3.3.3. The graph consisting of only one node labelled with the starting symbol S is called *the initial graph*. A graph is called a *terminal graph* if all its nodes are labelled with terminal symbols. The *language L(G) determined by a CFGG G* is the set of all terminal graphs that can be derived from the initial graph.

□

Because P is finite, there is a constant upper bound to the degree of the nodes of every g ∈ L(G). This implies already that not all classes of graphs are context-free. The class of *wheels*, where the n-th wheel consists of a circle of n nodes all connected to one centre node (see figure 4.3.3.4.), is an example of a non context-free class of graphs.

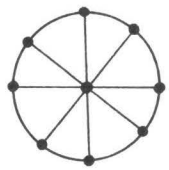


Figure 4.3.3.4. A wheel.

The following lemmas are more specific about the classes of graphs that are or are not context-free. If P and Q are graphs, PQ denotes a graph consisting of P and Q and a number of edges connecting P and Q.

Lemma 4.3.3.1(The *pumping lemma*). *Let* G *be a* CFGG. *If* L(G) *contains arbitrarily large (in terms of number of nodes) graphs, then* L(G) *contains graphs* TM^iO *for all* i=0,1,... *where* TM^i *is a star graph (see figure 4.3.3.5.).*

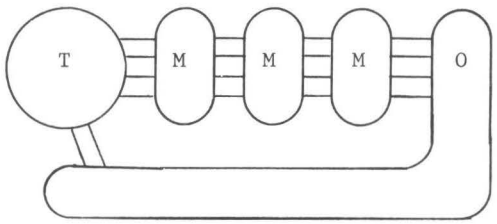
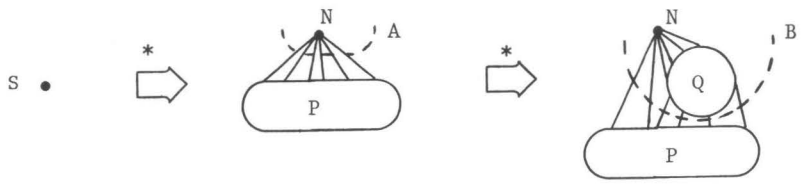


Figure 4.3.3.5. A graph TM^3O.

Proof. Because L(G) contains graphs of arbitrary size, there must be a non-terminal N producing itself:



where P and Q are subgraphs containing non-terminals and/or terminals. From N, P and Q we can generate subgraphs T, O, and M respectively, containing only terminal nodes. The star graph <NQ,B> is derived from <N,A> where B contains the same number of edges as A. The derivation of <NQ,B> from <N,A> can be re-

peated arbitrarily many times before the terminal subgraph T, O and M are generated yielding a graph TM^iO for any i.

□

Corollary 4.3.3.2. *If* L(G) *contains arbitrarily large graphs then there are constants* c *and* k *such that* L(G) *contains graphs of size* c+i.k *for all* i=0,1,...

□

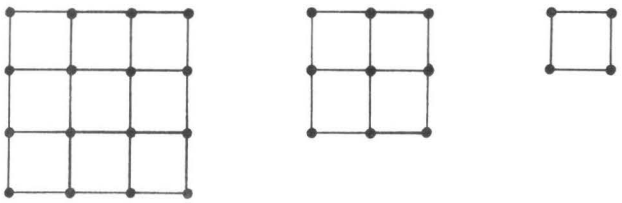


Figure 4.3.3.6. Some square grids.

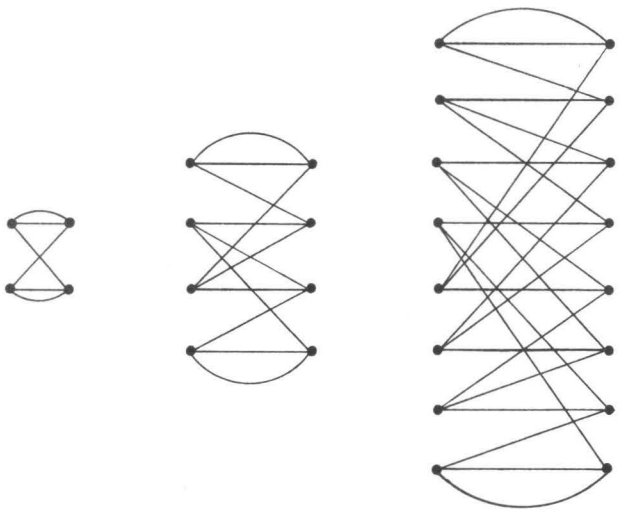


Figure 4.3.3.7. Some perfect shuffles.

It follows directly from this corollary that the class of square grids (see figure 4.3.3.6.) and the class of perfect shuffles are not context-free. A perfect shuffle is a bipartite graph LR, where both L and R contain $N=2^k$ nodes. The nodes from L and R are connected through an interlaced interconnection pattern (see figure 4.3.3.7.). In these cases we simply count the number of nodes in the members of the class (n^2 and 2^n, respectively). In most cases, however, this counting argument is too weak and we have to take the interconnection structure of the graphs into account.

Definition 4.3.3.4. A *(k,d)-reduction* of a graph is the substitution of a star subgraph (H,E) by a simple star (h,F) such that

(1) $|E| = |F|$ and F connects h to the same nodes of the rest of the graph as E did with H,

(2) H contains at most k nodes,

(3) all nodes of H have a degree at most d.

□

Clearly reduction is the inverse of derivation. Slisenko uses a similar notion, *contraction*, in order to prove that for every CFGG there is a polynomial-time algorithm for recognizing its language. The difference between reduction and contraction is that reduction is defined independently of a CFGG.

Definition 4.3.3.5. A graph is *(k,d)-reducible* iff it can be successively transformed into one node without edges by a sequence of (k,d)-reductions. A class of graphs is (k,d)-reducible iff all its members are.

□

If a graph is (k,d)-reducible, all stars in its reduction have at most d boundary edges. For example, the class of binary trees is (3,3)-reducible. Sufficient reductions are given in figure 4.3.3.8.

In the following we will ignore the labelling of graphs and consider their structure only.

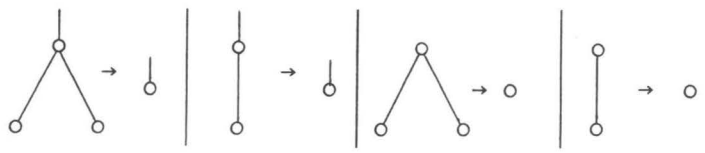


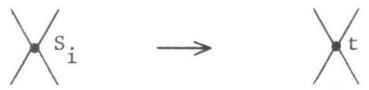
Figure 4.3.3.8. Reductions of a binary tree.

Lemma 4.3.3.3. *Consider a class of graphs* C. *There is a* CFGG G *such that* $C \subseteq L(G)$ *iff there are a* k *and* d *such that* C *is* (k,d)*-reducible.

Proof. Assume $C \subseteq L(G)$, for some CFGG G. For all $c \in C$ there is a derivation in G. All right-hand-sides of the productions are finite, so there are a k and d such that the number of nodes in each right-hand-side is at most k and all nodes are of degree at most d. A (k,d)-reduction is just the inverse of a derivation. (This part of the proof corresponds to a similar argument of Slisenko [77]).

Let C be (k,d)-reducible. For a given k and d there is only a finite number of stars St with at most k nodes of degree at most d and at most d boundary edges. Each of these stars will be used to form a production rule. There will be d+1 non-terminals $S_0,\ldots,S_d$ and one terminal t. S_i will be used only at nodes of degree i. For every star graph St with more than one node and i boundery edges we form the production rule:

Every node in St is labelled S_j iff it has the degree j. For every i we include a production rule:



Now let $G = \langle\{S_0,\ldots,S_d\},\{t\},P,S_0\rangle$ where P is the set of production rules defined above. Clearly every $c \in C$ can be derived using G, and $C \subseteq L(G)$.

□

Theorem 4.3.3.4(The *connectivity theorem*). *Let a set of graphs* S *contain arbitrarily large graphs. Let all subgraphs of* n *nodes of a graph in* S *of at least* 2n *nodes be connected to the rest of the graph by at least* F(n) *edges, where* F *is an increasing integer function. Then* S *is not (k,d)-reducible for any* k *and* d *(and therefore not a subset of a context-free graph language).*

Proof. Suppose on the contrary, that every $s \in S$ is (k,d)-reducible for some k and d. Choose an $s \in S$ with at least 2n nodes such that $n>k^2$ and $F(\frac{n}{k})>d$. A reduction of s is a sequence of graphs $s=s_0,s_1,\ldots,s_m$ where s_i is the result of a (k,d)-reduction of s_{i-1} and s_m is a single node. Every node in every s_i has a degree less than d. With every node x in an s_i we associate a number determining from how many original nodes in s_0 x has been reduced. Consider the sequence $M_0,M_1,\ldots,M_m$ where M_i = max{associated number of an x in s_i}. Clearly $M_0=1$ and $M_m \geq 2n$. The sequence is non-decreasing and $M_i \leq k.M_{i-1}$. So there is a p such that $M_{p-1} \leq \frac{n}{k}$ and $\frac{n}{k} < M_p \leq n$ (because $n>k^2$). Let M_p be the associated number of a node Y in s_p. Y has a degree d_Y equal to the number of edges that connects the subgraph consisting of the M_p original nodes of Y in s_0 and all their internal edges to the rest of the graph, so $d_Y \geq F(M_p) \geq F(\frac{n}{k})$. But we have chosen s such that $F(\frac{n}{k})>d$ which contradicts the supposition.

□

In a square grid of at least 2n nodes all subgraphs of k<n nodes are connected to the rest of the graph by more than $\sqrt{k}$ edges so any set of graphs containing square grids (such as the class of all grids) is not context-free.

If one wants to generate graphs with high connectivity such as grids or shuffles, a more powerful kind of expansion is needed. Two extensions of the expand statement are considered. The first is to allow the declaration of *channel arrays* combined with a loop construct in an expand statement such that any graph can be generated in one expansion. Generating a perfect shuffle can be written as in figure 4.3.3.9.

```
n = 2^k; /* k≥1 */
expand chan c1 [0..2n-1], c2[0.. n-1]
       for i in [0..n-1]
        do create l(in c2[i] out c1[2i], c1[2i + 1])
           create r(in c1[i], c1[n + i] out c2[i])
        od
endexp
```

Figure 4.3.3.9. Generating a shuffle iteratively.

A second possible extension is to allow arrays of channels as formal parameters in process declarations and as actual parameters in creations. The recursive nature of the graphs one wants to generate can then be expressed elegantly. This kind of graph expansion is a generalization of *separators* as defined be Hoey and Leiserson [41]. A separator is defined as follows. A *bisection* S of a graph G=(V,E) into graphs G'=(V',E') and G"=(V",E") is a disjoint partition of nodes V=V'U V" together with a disjoint partition of the edges E=E'U E"U E_s, such that |V'| and |V"| differ at most one. $|E_s|$ is called the *bisection width* of S. A *separator* for a class of graphs is a set of bisections, at least one for every graph in the class. The generation of a perfect shuffle is now written as in figure 4.3.3.10. The two *wings* making up a shuffle are shown in figure 4.3.3.11.

```
begin
    n = 2^k ; /* k ≥ 1 */
    if n ≥ 4 then m = 2^(k-1); l = 2^(k-2);
                  expand chan c[0..n-1]
                         create wing (in c[0..l-1], c[m..3l-1]
                                      out c[0..m-1])
                         create wing (in c[l..m-1], c[3l..n-1]
                                      out c[m..n-1]
                  endexp
             else expand chan c[0..1]
                         create wing (in c[0], c[1] out c[0..1])
                  endexp
    fi
end
process wing ( in c1[0..l], c2[0..m] out c3[0..n]):
begin
    if l>0 then expand create wing (in c1[0..l/2], c2[0..m/2]
                                    out c3[0..n/2])
                       create wing (in c1[l/2+1..l], c2[m/2+1..m]
                                    out c3[n/2+1..n])
                endexp
           else expand chan i[0..3]
                       create left(in c1[0] out i[0..1])
                       create right(in i[0], i[2] out c3[0])
                       create left(in c2[0] out i[2..3])
                       create right(in i[1], i[3] out c3[1])
                endexp
    fi
end
```

Figure 4.3.3.10. Generating a perfect shuffle recursively.

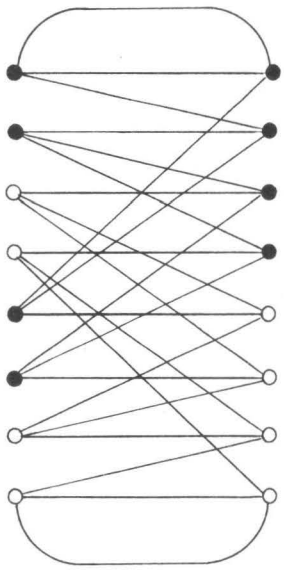


Figure 4.3.3.11. A perfect shuffle viewed as two wings.

4.4. SOME DEFINITIONS AND THEOREMS FROM THE THEORY OF NP-COMPLETENESS

To characterize the computional power of DNP programs in general we will need the following definitions and theorems from the theory of NP-completeness [32].

A *problem* Π is a set of *instances* of a question. Take as an example the problem PRIMES, an element of which is : is *1234567* a prime number? We will consider problems that can be posed as *decision problems*, the instances having two possible answers: yes or no.

Definition 4.4.1. An *encoding scheme* e for a problem Π provides a way to describe each instance of Π by an appropriate string of symbols over some alphabet Σ. Π and e partition Σ^* into three classes of strings:

(1) those strings that do not encode an instance of Π,
(2) those strings that encode a yes-instance of Π,
(3) those strings that encode a no-instance of Π.

The language associated with Π and e, $L(\Pi,e)$, is the class of strings encoding yes-instances of Π.

□

Definition 4.4.2. Consider a deterministic Turing machine (DTM) or non-deterministic Turing machine (NDTM) M, reading strings over Σ and having two halt states q_y and q_n. The language L_M accepted by M is the set of input strings $x \in \Sigma^*$ for which (one of the computations of) M halts in q_y.

□

Definition 4.4.3. A Turing machine M *solves* a decision problem Π under encoding e iff $L_M = L(\Pi,e)$.

□

Definition 4.4.4. $P = \{L$: there is a polynomial time bounded DTM M such that $L = L_M\}$. In other words, Π belongs to P under encoding e if there is a polynomial time DTM solving Π under e.

□

Definition 4.4.5. $NP = \{L$: there is a polynomial time bounded NDTM M such that $L = L_M\}$.

□

It is open whether P and NP are equal (the P = NP problem, see [18]).

Definition 4.4.6. A *polynomial transformation* (or *p-reduction*) from a language $L_1 \subseteq \Sigma_1^*$ to a language $L_2 \subseteq \Sigma_2^*$ is a function $f: \Sigma_1^* \to \Sigma_2^*$ such that

(1) there is a polynomial time DTM computing f, and

(2) for all $x \in \Sigma_1^*$: $x \in L_1$ iff $f(x) \in L_2$.

If there is a p-reduction from L_1 to L_2 we write $L_1 \propto L_2$.

□

Definition 4.4.7. $L \in NPC$ iff $L \in NP$ and for all $L' \in NP$: $L' \propto L$.
In other words, a decision problem is *NP-complete* if it is in NP and all NP problems can be polynomially transformed to it.

□

Assuming $P \neq NP$, the world of NP can be pictured as in figure 4.4.1.

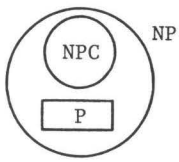


Figure 4.4.1. The world of NP.

The *satisfiability problem* (SAT) can be stated as follows. We have a set of boolean variables U, a subset B of the 16 possible binary boolean operators and a well-formed boolean expression E over U and B. The question is whether there is a truth assignment for U satisfying E.

Theorem 4.4.1. SAT *is NP-complete.* ([18])

□

Definition 4.4.8. PSPACE = {L: there is a polynomial space bounded DTM M such that $L = L_M$}.

□

Definition 4.4.9. $L \in$ PSPACEC iff $L \in$ PSPACE and for all $L' \in$ PSPACE: $L' \propto L$. In other words, Π is PSPACE-complete if it belongs to PSPACE and all PSPACE problems are p-reducible to Π.

□

Clearly P $\subseteq$ PSPACE and NP $\subseteq$ PSPACE.

The *quantified boolean formulas problem* (QBF) can be stated as follows. We have a well-formed quantified boolean formula $F = (Q_1x_1)(Q_2x_2)..(Q_nx_n)E$, where E is a boolean expression involving variables $x_1,...,x_n$ and each Q_i is one of the quantifiers $\exists$ and $\forall$. The question is whether there is a truth assignment for $x_1,...,x_n$ satisfying F.

Theorem 4.4.2. QBF *is PSPACE-complete.*

□

Further details and proofs can be found in the book of Garey and Johnson [32].

4.5. DNP-PROGRAMS FOR NP-COMPLETE AND PSPACE-COMPLETE PROBLEMS

Unless somebody proves that P=NP after all, there seems to be no better way to tackle NP-complete problems than by trial and error. A trial and error algorithm consists of two stages, the first being a guessing stage and the second being a checking stage. Both guessing and checking of one solution can be done in polynomial time for NP-complete problems, but the number of possible guesses is exponential in the length of the instance of the problem. On a sequential machine this trial and error technique has therefore an exponential worst case time complexity.

A useful property of the trial and error technique is that the various checks are independent of each other. This makes the technique suitable for a parallel implementation. Generally speaking, in a parallel implementation the guesses are issued in polynomial time, all guesses are checked by independent processes simultaneously, and the answers are combined again in polynomial time. Implementations of particular problems may be clever by pruning the tree of all guesses. The scheme is very similar to the divide-and-conquer algorithms of section 4.2.3.

Proposition 4.5.1. *A complete binary tree of* 2^n-1 *processes can be generated in* $O(n)$ *time-steps using the tree expansion of figure 4.5.1.*

□

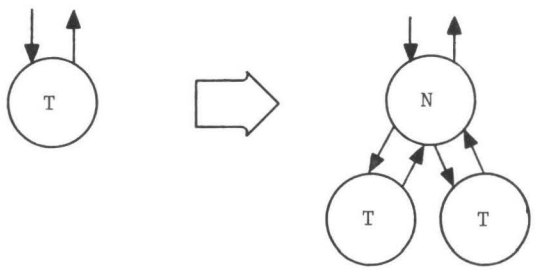


Figure 4.5.1. Tree expansion.

We say that a DNP program *accepts* an instance of a problem Π under encoding e, if upon receipt of the encoded instance the program outputs "YES".

Definition 4.5.1. P_{DNP} = {L: there is a polynomial time DNP program accepting L}.

Theorem 4.5.2. $NP \subseteq P_{DNP}$.

Proof. The root process goes through the input and transforms it to an instance of SAT, which takes polynomial time P. This yields a boolean expression E. If E contains no variables, it will be evaluated and the result will be output. Otherwise, a variable v in E is selected and two expressions E_T

and E_F are generated where E_T is E with TRUE substituted for v and E_F is E with FALSE substituted for v. The root now expands as suggested in proposition 4.5.1. The left child will deal recursively with E_T, the right child with E_F. The root will combine the answers. Let the length of E be n and let E contain m variables. Both m and n are less than P. The resulting tree will be at most m layers deep. Evaluating E, selecting v and generating E_T and E_F takes O(length of E) steps. Therefore the whole algorithm takes O(m.n+P) steps.

□

Theorem 4.5.3. $PSPACE \subseteq P_{DNP}$.

Proof. The root process goes through the input and transforms it to an instance of QBF, (see section 4.4.), which takes polynomial time. This yields a quantified boolean formula E. The root handles x_1. It generates two expressions E_T and E_F from E just as in theorem 4.5.2. It expands, the left child deals with $(Q_2x_2) \ldots (Q_nx_n)E_T$, the right child with $(Q_2x_2) \ldots (Q_nx_n)E_F$, and afterwards the root combines the answers according to the quantifier Q_1. The same reasoning as in theorem 4.5.2. shows that the program takes polynomial time.

□

4.6. DNP PROGRAMS AND N-RAMS

An N-RAM (Wyllie [90]) consists of an unbounded set of processors $P_0,P_1,\ldots$, an unbounded set of communication links $C_0,C_1,\ldots$, a set of input registers and a finite program. Each processor has an accumulator, an unbounded local memory, a program counter, and a flag indicating whether or not the processor is running. All memory locations are capable of holding non-negative integers. A program consists of a sequence of possibly labelled instructions chosen from the list in figure 4.6.1.

Initially the input to the N-RAM is placed in the input registers, all memory is cleared, the length of the input is placed in the accumulator of P_0, and P_0 is started at the first instruction of the program.

Instruction		Meaning
LOAD	operand	Transfer to/from accumulator from/to local memory.
STORE	operand	
ADD	operand	Add/substract the value of operand to/from the accumulator.
SUB	operand	
JUMP	label	Branch/branch on zero-accumulator to label.
JZERO	label	
READ	operand	Place contents of specified input register in accumulator.
FORK	label	see text.
HALT		see text.
SEND	operand	see text.
RECEIVE	operand	see text.

Figure 4.6.1. The N-RAM instruction set.

A program is non-deterministic if some label occurs more than once, deterministic otherwise. Each operand may be a literal, and address or an indirect address. Execution is synchronous. At each step each running processor executes the instruction given by its program counter. A *FORK label* instruction executed by processor P_i selects an inactive processor P_j, clears P_j-s local memory, copies P_i-s accumulator into P_j-s accumulator and starts running at *label*. A *HALT* instruction causes a processor to stop running.

For a word to be sent from one processor to another, one processor must execute a *SEND operand* instruction while the other simultaneously executes a *RECEIVE operand* instruction. The parameters to SEND and RECEIVE specify one of the possible communication links. An unmatched SEND or RECEIVE behaves as a null instruction. The accumulator serves as source and target for the value to be transmitted. Execution of a FORK instruction causes the father and child process to be connected by a communication link, the number of which is available to both. This enables the father process to send necessary informa-

tion , for example about the communication links the child is allowed to use. Wyllie shows that new communication links can be allocated without giving rise to conflicts, i.e. a communication link will never be used by more than one sender and one receiver simultaneously.

We want to show that N-RAMs can simulate DNP-programs within a polynomial time factor. Because N-RAMs cannot multiply in one unit of time and because we want the N-RAMs to model the parallelism in DNP-programs we restrict the operators in the expressions in ordinary DNP-statements to additions and subtractions. We call such programs *restricted DNP programs*. (Note that the DNP-programs for NP- and PSPACE-problems constructed in section 4.5. are all restricted in this sense.)

Definition 4.6.1. $P_{\text{Restricted-DNP}}$ = {L: there is a polynomial time restricted DNP program accepting L}.

Theorem 4.6.1. $P_{Restricted\text{-}DNP} \subseteq P_{N\text{-}RAM}$.

Proof. Translating DNP-programs to N-RAMs is very similar to translating DNP to UNIX as presented in chapter three. A difference is that N-RAM processors communicate instantaneously. We therefore allocate for a DNP-channel an N-RAM processor, whose local memory will contain the queue of tokens. These channel processors are used just as the UNIX pipes. They are represented by two communication links, one for inserting and one for deleting tokens. Simulating expansion by means of repeated forking is also done as in the UNIX implementation of DNP. The actual channel information needed by a newly created process will be passed over the communication links connecting the FORKer and the FORKed. The process to be run in the new processor is represented by the label in the FORK instruction. The DNP-program and its N-RAM simulation differ only by a polynomial factor in running time.

□

Because $P_{\text{N-RAM}}$ = PSPACE [90] we can conclude the following.

Theorem 4.6.2. $P_{Restricted\text{-}DNP} \subseteq PSPACE$.

□

Theorem 4.6.3. $P_{Restricted\text{-}DNP} = PSPACE$.

Proof. In the proof of theorem 4.5.3. that PSPACE $\subseteq P_{DNP}$, we did not rely on multiplication in one unit of time, so we can conclude:
PSPACE $\subseteq P_{Restricted\text{-}DNP}$. Together with theorem 4.6.2. this implies:
$P_{Restricted\text{-}DNP}$ = PSPACE.
□

CHAPTER FIVE

THE CORRECTNESS OF DNP PROGRAMS

5.1. INTRODUCTION

In this chapter we develop correctness proofs for some of the programs from chapter four. The proofs are based on a semantics of DNP based on the work of Kahn [46]. For the sake of completeness we present an informal introduction to this semantics.

A process takes its input values one by one from its input channels. Its actions are completely deterministic. If a process terminates it writes a special *end-of-file* mark (EOF) on all its output channels. The last value on an input channel of the net will also be EOF.

A process specifies a function which takes *input histories* as arguments and yields *output histories* as values. A history models the sequence of values that travelled over a channel from the beginning of the computation until a certain moment. Histories can be ordered according to the amount of information they contain. History Y contains more information than history X, written $X \subseteq Y$, iff X is a prefix of Y.

The history functions defined by processes have a number of important properties. If input history X is a prefix of input history Y, the process will act identically on the common prefix and will thus generate the same values on the output channel. The remaining input on Y can only have the effect of writing more values on the output channel. In formula: if $X \subseteq Y$ then $f(X) \subseteq f(Y)$ where f is the function describing the behaviour of the process. This property is called *monotonicity*. In Kahn's model all processes compute monotonic history functions.

A second important property of the history functions is *continuity*, which concerns the approximation of an infinite sequence by its finite prefixes. The prefixes of a history X form a *chain*, i.e., a sequence of histories $X_1, X_2, ..$ such that $X_i \subseteq X_{i+1}$ for every $i \geq 1$.

Lemma 5.1.1[46]. *Every chain has a least upper bound* UX_i.

Proof. Either the chain is stable, i.e., there is a k such that $X_k = X_{k+1} = ..$ and $UX_i = X_k$, or the chain is not stable. But then every element X_i has a successor of length greater than X_i and UX_i will be the infinite history X with the property that all X_i are a prefix of X.

□

A one-input-channel one-output-channel process P yields an output history f(X) when given an input history X, where f is the function associated with P. Consider a chain $X_1, X_2, ..$ with $UX_i = X$. Because f is monotonic the values $f(X_1), f(X_2), ..$ also form a chain. This means that an arbitrary finite approximation of f(X) is obtained by letting P work on a finite input history $X_k \subseteq X$, i.e., for every element Y of f(X) only a finite number of elements of X have been read at the moment it is generated. Furthermore, the whole sequence up to Y has then been generated. This is equivalent to saying that $f(X) = f(UX_i) = Uf(X_i)$. This property is called *continuity*. In Kahn's words: continuity prevents the possibility of a process deciding to send some output only after it has received an infinite amount of input.

In [46] it is stated that the function describing the meaning of a process can be obtained "by the usual method of McCarthy for converting flowchart programs to recursive definitions". It is not clear how to apply this method to processes containing expand statements even though the semantics of an expand statement is not much different from that of a series of procedure calls. What is needed is a formal semantics of the language. Such a formal semantics for a syntactically simplified version of DNP is given in [14], where an operator is defined that (i) takes a process-declaration and translates it into a function from input-histories to output-histories and (ii) takes a DNP program (a sequence of process-declarations and a main body) and translates it into a set of equations. To each channel of the (initial) net of the program a variable is associated. For all variables X associated

with input channels to the net there is an equation X=I where I is an input history. For each process with n input channels $X_1,\ldots,X_n$ and m output channels $Y_1,\ldots,Y_m$ there are m equations $Y_i = f_i(X_1,\ldots,X_n)$ where f_i is the function that describes the behaviour of the process as far as the i-th output channel is concerned. The meaning of a DNP program is defined as the minimal solution to the set of equations.

Theorem 5.1.2[46]. *The set of equations describing the meaning of a network admits a unique minimal solution. Executing the program results in a set of histories described by the minimal solution.*

□

We will now discuss how a function is derived from a process declaration. A formal treatment can be found in [14]. The process heading of a process declaration determines the number of input and output parameters of the function to be derived. If, for example, a process heading has two input channels and three output channels the associated function will have the form f(X,Y) = (P,Q,R). The output that a process yields is generally not only dependent on the input histories but also on the value parameters and the values of the local variables. The associated function will therefore often have extra parameters giving the relevant part of the internal state of the process.

A process body is a sequence of statements $S_1;S_2;\ldots;S_n$. The associated function can be derived stepwise by *concatenating* the *effect of* S_1 to a function f' describing the *effect of* $S_2,\ldots,S_n$. In the above example:

$$f(X,Y,Z) = (<p>,<q>,<r>)\hat{}f'(X',Y',Z')$$

where <p>,<q>, and <r> stand for the sequences of values written by S_1 on P, Q and R, ^ denotes simultaneous concatenation defined as $(X_1,\ldots,X_n)\hat{}(Y_1,\ldots,Y_n) = (X_1\hat{}Y_1,\ldots,X_n\hat{}Y_n)$, X' and Y' stand for the input histories that may have changed because of S_1, and Z' denotes the changed internal state Z.

The above expression is rather general. We will be more concrete and take for S_1 a read statement, a write statement, an assignment, a conditional statement, a loop, and an expand statement respectively. Where needed we will add internal state-parameters to f'.

If S_1 is *read(X,x)*, the associated function is f(X,Y) = f'(R(X),Y,F(X)).

F(X) stands for the first element of X. F(X) has become part of the internal state. R(X) stands for the rest of X, f' denotes the effect of the rest of the process.

If S_1 is *write(P,p)*, the associated function is f(X,Y) = (<p>,<>,<>)^f'(X,Y).

If S_1 is $x = e$, the associated function is f(X,Y) = f'(X,Y,e).

If S_1 is *if(b)* $\{S_2\}$ *else* $\{S_3\}$, the associated function is

```
f(X,Y) = [b → (<p>,<q>,<r>)^f'(X',Y')
           , (<p'>,<q'>,<r'>)^f"(X",Y")
         ].
```

The construct [A → B,C] denotes the conditional function. Depending on the truth value of A either B or C applies.

If S_1 is *while(b)* $\{S_2\}$, the associated function is

```
f(X,Y) = [b → (<p>,<q>,<r>)^f(X',Y')
           , f'(X,Y)].
```

If S_1 is an expand statement, the effect of S_1 is defined as the effect of the network into which it expands, i.e., the solution of a set of equations derived from the network. The right hand sides of these equations have the form $g(Z_1,\ldots,Z_k)$ and the g-s are specified by either a creation or a survival. The function corresponding to a creation is defined (recursively) by a process declaration. The function corresponding to a survival will be derived from the rest of the process declaration, i.e., from the statements $S_2,\ldots,S_n$. So if S_1 is

```
expand chan C
   create filter (in X out C)
   keep me (in C,Y out P,Q,R)
endexp
```

the associated function is f(X,Y) = f'(C,Y) where $C = f_{filter}(X)$.

In subsequent sections we will prove properties of programs by first translating a program into a set of equations and then solving these equations.

5.2. CORRECTNESS OF PIPELINE SORT WITH SINGLE NUMBERS INTERNALLY

Consider the sort program from figure 4.2.1.7. Figure 5.2.1. shows the initial network.

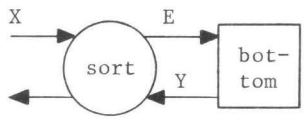


Figure 5.2.1. Initial sorting network.

The two processes are described by the functions f_{sort} and f_{bottom} for which the following holds:

$$f_{bottom}(E) = \langle EOF \rangle$$

$$f_{sort}(X,Y) = [F(X) = EOF \rightarrow (Y,\langle EOF \rangle) \\ , f_{sort\text{-}merge}(R(X),Y,F(X))]$$

The function f_{sort} reflects the test whether there are (still) elements to be sorted and the actions taken upon that test; $f_{sort\text{-}merge}$ describes the action taken when there are elements to be sorted: the net expands as shown in figure 5.2.2.

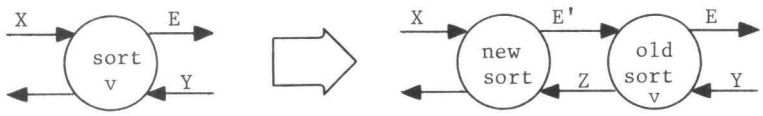


Figure 5.2.2. Expansion of sort.

$$f_{sort\text{-}merge}(X,Y,v) = (f_{sort}(X,f_{merge}(E',Y,v)\downarrow 1)\downarrow 1, f_{merge}(E',Y,v)\downarrow 2) \\ \text{where } E' = f_{sort}(X,f_{merge}(E',Y,v)\downarrow 1)\downarrow 2$$

$$f_{merge}(X,Y,v) = [F(Y) = EOF \rightarrow (\langle v,EOF \rangle,\langle EOF \rangle) \\ , (\min(F(Y),v),\langle\rangle)\hat{}f_{merge}(X,R(Y),\max(F(Y),v))]$$

The functions min and max yield the minimum and maximum of their arguments, respectively. The $\downarrow$i-operator is defined as $(X_1,\ldots,X_n)\downarrow i = X_i$ $(1 \leq i \leq n)$.

Now we can write down the equations which denote the meaning of the program:

$$(\text{sorted},E) = f_{\text{sort}}(X,Y)$$

$$Y = f_{\text{bottom}}(E)$$

which are transformed straightforwardly into:

$$(\text{sorted},E) = f_{\text{sort}}(X,\langle\text{EOF}\rangle)$$

In proving that $f_{\text{sort}}(X,\langle\text{EOF}\rangle)\downarrow 1$ is an ordering of X we use the following lemmas.

Lemma 5.2.1(Behaviour of f_{merge}). *If* Y *is a finite and ordered sequence of numbers and* v *is an arbitrary number, then* $f_{\text{merge}}(X,Y',v)\downarrow 1$ *is an ordered permutation of* Y^<v>, *followed by* <EOF>, *where* Y' = Y^<EOF>.

Proof. By induction on the length of Y.

Base: |Y| = 0: $f_{\text{merge}}(X,Y',v)\downarrow 1 = \langle v,\text{EOF}\rangle$

Step: |Y| = k>0: $f_{\text{merge}}(X,Y',v) = \langle \min(F(Y'),v)\rangle \hat{}\, f_{\text{merge}}(X,R(Y'),\max(F(Y'),v))\downarrow 1$

We have:

(1) $\min(F(Y'),v) \leq \max(F(Y'),v)$

(2) $\min(F(Y'),v) \leq x$, for all $x \in R(Y')$

From (1), (2) and the induction hypothesis we conclude that $f_{\text{merge}}(X,Y',v)\downarrow 1$ is an ordered permutation of Y^<v> followed by <EOF>.

□

Lemma 5.2.2(Behaviour of $f_{\text{sort-merge}}$). *If* X *is a finite sequence of numbers,* Y *is a finite and ordered sequence of numbers, and* v *an arbitrary number, then* $f_{\text{sort-merge}}(X',Y',v)\downarrow 1$ *is an ordered permutation of* X^Y^<v> *followed by* <EOF>, *where* X' = X^<EOF> *and* Y' = Y^<EOF>.

Proof. By induction on the length of X.

Base: |X| = 0: $f_{sort-merge}(X',Y',v)\downarrow 1 = f_{sort}(X',f_{merge}(E',Y',v)\downarrow 1)\downarrow 1 =$ $f_{merge}(E',Y',v)\downarrow 1 = Z\hat{}\langle EOF\rangle$ where Z is an ordered permutation of $Y\hat{}\langle v\rangle$ (previous lemma)

Step: |X| = k>0: $f_{sort-merge}(X',Y',v)\downarrow 1 = f_{sort}(X',f_{merge}(E',Y',v)\downarrow 1)\downarrow 1 =$ $f_{sort-merge}(R(X'),Z\hat{}\langle EOF\rangle,F(X'))\downarrow 1$

(1) Z is an ordered permutation of $Y\hat{}\langle v\rangle$ (previous lemma)

According to (1) and the induction hypothesis we can conclude that $f_{sort-merge}(X',Y',v)\downarrow 1$ is an ordered permutation of $X\hat{}Y\hat{}\langle v\rangle$, followed by <EOF>.

□

Theorem 5.2.3(Behaviour of the sorting program). *If* X *is a finite sequence of numbers, then* $f_{sort}(X\hat{}\langle EOF\rangle,\langle EOF\rangle)\downarrow 1$ *is an ordered permutation of* X, *followed by* <EOF>.

Proof. By definition of f_{sort} and lemma 5.2.2.

□

The correctness proof of pipeline sort with deques internally (figure 4.2.1.6) is slightly more complicated [10]. The third parameter of f_{merge} becomes a finite sequence of numbers playing the role of the *deque d* in *process sort*. The proof extends in that we have to show that *deque d* stays ordered and that given an ordered deque and an ordered input sequence f_{merge} yields an ordered output sequence.

5.3. CORRECTNESS OF MATMUL

Consider the program Matmul (figure 4.2.2.6.) for multiplying two square matrices. The data travelling over the channels has a prescribed format: a sequence of sequences of integers, separated and possibly preceeded by SEP-tokens ($) and terminated by an EOM-token (*). The program does not check whether the input matrices have the correct format, nor whether the input matrices have the same size. The main process madm determines whether the input matrices are empty (<*>) and if not, expands into the initial network shown in figure 5.3.1.

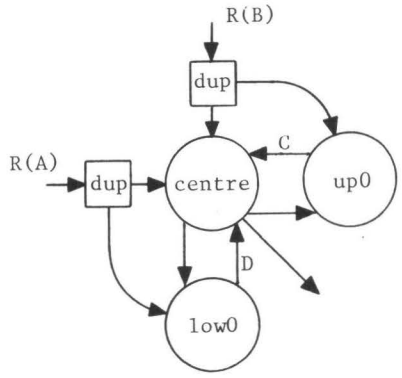


Figure 5.3.1. Initial network of Matmul.

The function describing madm is:

$$f_{madm}(A,B) = [F(A) = * \rightarrow <*>$$
$$, f_{centre}(f_{dup}(R(A))\downarrow 1, f_{dup}(R(B))\downarrow 1, C, D)\downarrow 3]$$

where $C = f_{up0}(f_{dup}(R(B))\downarrow 2, f_{centre}(f_{dup}(R(A))\downarrow 1, f_{dup}(R(B))\downarrow 1, C, D)\downarrow 1)$

and $D = f_{low0}(f_{dup}(R(A))\downarrow 2, f_{centre}(f_{dup}(R(A))\downarrow 1, f_{dup}(R(B))\downarrow 1, C, D)\downarrow 2)$

$$f_{centre}(A,B,C,D) = (<\$>,<\$>,<>)\hat{}f_{centre'}(A,B,C,D,0,<>)$$

$f_{centre'}$ has four input history parameters and two value parameters. The first two histories model the input rows and columns, the third and fourth model the upper and lower triangles of the product matrix which are computed by up0 and low0 respectively. The two value parameters model the relevant part of the internal state of centre: the first parameter plays the role of the variable x, the second parameter plays the role of the variable q. Consequently, the second parameter takes the form of a sequence. $f_{collect}$ describes the collecting phase of centre.

$$f_{centre'}(A,B,C,D,x,q) =$$
$$[F(A) = * \rightarrow (<*>,<*>,<>)\hat{}f_{collect}(R(A),R(B),C,D,q\hat{}<x>),$$
$$[F(A) = \$ \rightarrow (<\$>,<\$>,<>)\hat{}f_{centre'}(R(A),R(B),C,D,0,q\hat{}<x>),$$
$$(<F(A)>,<F(B)>,<>)\hat{}f_{centre'}(R(A),R(B),C,D,x+F(A).F(B),q)]]$$

$$f_{collect}(A,B,C,D,q) =$$
$$[F(C) = * \rightarrow (\diamond,\diamond,<\$,F(q),*>)$$
$$, (\diamond,\diamond,<\$,F(q)>)\hat{}f_{collectup}(A,B,R(C),R(D),R(q))]$$

$$f_{collectup}(A,B,C,D,q) =$$
$$[F(C) = * \rightarrow (\diamond,\diamond,<\$>)\hat{}f_{collectlow}(A,B,R(C),D,q),$$
$$[F(C) = \$ \rightarrow (\diamond,\diamond,<\$>)\hat{}f_{collectlow}(A,B,R(C),D,q),$$
$$(\diamond,\diamond,<F(C)>)\hat{}f_{collectup}(A,B,R(C),D,q)]]$$

$$f_{collectlow}(A,B,C,D,q) =$$
$$[F(D) = * \rightarrow (\diamond,\diamond,<F(q),*>),$$
$$[F(D) = \$ \rightarrow (\diamond,\diamond,<F(q)>)\hat{}f_{collectup}(A,B,C,R(D),R(q)),$$
$$(\diamond,\diamond,<F(D)>)\hat{}f_{collectlow}(A,B,C,R(D),q)]]$$

An up0 process determines by reading a column item whether it will have to compute a diagonal. If so it expands as shown in figure 5.3.2. We now get a similar argument for up0 and low0 as for centre.

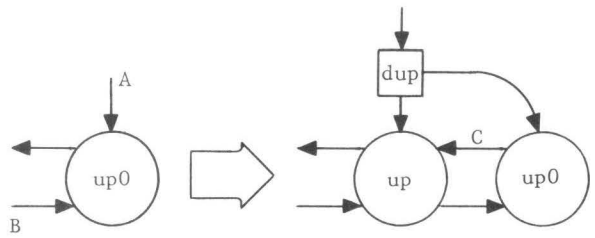


Figure 5.3.2. Expansion of up0.

$$f_{up0}(A,B) = [F(A) = * \rightarrow <*>$$
$$, f_{up}(f_{dup}(R(A))\downarrow 1,R(B),C)\downarrow 2]$$
$$\text{where } C = f_{up0}(f_{dup}(R(A))\downarrow 2, f_{up}(f_{dup}(R(A))\downarrow 1,R(B),C)\downarrow 1)$$

$$f_{up}(A,B,C) = (<\$>,\diamond)\hat{}f_{up},(A,B,C,0,\diamond)$$

$$f_{up'}(A,B,C,x,q) =$$
$$[F(A) = * \rightarrow (<*>,<>)\hat{}f_{uptriangle}(R(A),R(B),C,q\hat{}<x>),$$
$$[F(A) = \$ \rightarrow (<\$>,<>)\hat{}f_{up'}(R(A),R(B),C,0,q\hat{}<x>),$$
$$(F(B),<>)\hat{}f_{up'}(R(A),R(B),C,x+F(A).F(B),q)]]$$

$$f_{uptriangle}(A,B,C,q) =$$
$$[F(C) = * \rightarrow (<>,<\$,F(q),*>),$$
$$[F(C) = \$ \rightarrow (<>,<\$,F(q)>)\hat{}f_{uptriangle}(A,B,R(C),R(q)),$$
$$(<>,<F(C)>)\hat{}f_{uptriangle}(A,B,R(C),q)]]$$

A low0 process determines whether it will have to compute a diagonal. If so, it expands as shown in figure 5.3.3.

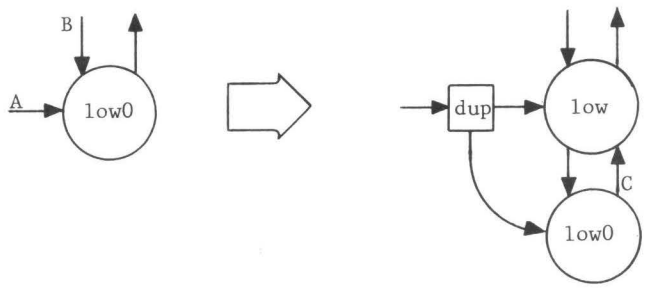


Figure 5.3.3. Expansion of low0.

$$f_{low0}(A,B) = [F(A) = * \rightarrow <*>$$
$$, f_{low}(f_{dup}(R(A))\downarrow 1,R(B),C)\downarrow 2]$$
$$\text{where } C = f_{low0}(f_{dup}(R(A))\downarrow 2,f_{low}(f_{dup}(R(A))\downarrow 1,R(B),C)\downarrow 1)$$

$$f_{low}(A,B,C) = (<\$>,<>)\hat{}f_{low'}(A,B,C,0,<>)$$

$$f_{low'}(A,B,C,x,q) =$$
$$[F(A) = * \rightarrow (<*>,<\$>)\hat{}f_{lowtriangle}(R(A),B,C,q\hat{}<x>),$$
$$[F(A) = \$ \rightarrow (<\$>,<>)\hat{}f_{low'}(R(A),R(B),C,0,q\hat{}<x>),$$
$$(<F(B)>,<>)\hat{}f_{low'}(R(A),R(B),C,x+F(A).F(B),q)]]$$

$$f_{lowtriangle}(A,B,C,q) =$$
$$[F(C) = * \rightarrow (\langle\rangle,\langle F(q),*\rangle),$$
$$[F(C) = \$ \rightarrow (\langle\rangle,\langle F(q),\$\rangle)\hat{}f_{lowtriangle}(A,B,R(C),R(q)),$$
$$(\langle\rangle,\langle F(C)\rangle)\hat{}f_{lowtriangle}(A,B,R(C),q)]]$$

The function describing dup is:

$$f_{dup}(A) = [F(A) = * \rightarrow (\langle *\rangle,\langle *\rangle),$$
$$[F(A) = \$ \rightarrow (\langle \$\rangle,\langle \$\rangle)\hat{}f_{dup}{}'(R(A)),$$
$$(\langle F(A)\rangle,\langle\rangle)\hat{}f_{dup}(R(A))]]$$

$$f_{dup}{}'(A) = [F(A) = * \rightarrow (\langle *\rangle,\langle *\rangle)$$
$$, (\langle F(A)\rangle,\langle F(A)\rangle)\hat{}f_{dup}{}'(R(A))]$$

The equation denoting the meaning of the program is:

$$product = f_{madm}(Mr,Mc)$$

As before we state some lemmas that will be used to prove the main theorem. In these lemmas we will use a special notation for the format of the input and output sequences.

Notation. Suppose $R_1,\ldots,R_n$ are finite sequences of integers. Then

- $[R_i,\ldots,R_{i-1}] = [] = \langle *\rangle$
- $[R_1,\ldots,R_n] = \langle \$\rangle\hat{}R_1\hat{}[R_2,\ldots,R_n]$

Furthermore $\{R_1,\ldots,R_n] = R_1\hat{}[R_2,\ldots,R_n]$ $(n>0)$

Lemma 5.3.1(Behaviour of f_{dup}).
Let $n\geq 1$ *and* $A = \{A_1,\ldots,A_n]\hat{}X$.
Then $f_{dup}(A) = (\{A_1,\ldots,A_n],[A_2,\ldots,A_n])$.

Proof. I: $f_{dup}(A) = (A_1,\langle\rangle)\hat{}f_{dup}([A_2,\ldots,A_n])$ by induction on the length of A_1.

II: $f_{dup}{}'(\{A_i,\ldots,A_n]) = (A_i,A_i)\hat{}f_{dup}{}'([A_{i+1},\ldots,A_n])$ $(i\leq n)$ by induction on the length of A_i.

III: The lemma now follows by induction on n.

□

Lemma 5.3.2(Behaviour of $f_{uptriangle}$).

Let $k \geq 0$.

Then $f_{uptriangle}(A,B,[C_1,\ldots,C_k],\langle x_1,\ldots,x_{k+1}\rangle) =$

$(\langle\rangle,[\langle x_1\rangle\hat{}C_1,\ldots,\langle x_k\rangle\hat{}C_k,\langle x_{k+1}\rangle])$.

Proof. I: If $k \geq 1$ $f_{uptriangle}(A,B,[C_1,\ldots,C_k],\langle x_1,\ldots,x_{k+1}\rangle) =$

$(\langle\rangle,\langle \$,x_1\rangle)\hat{}f_{uptriangle}(A,B,\{C_1,\ldots,C_k],\langle x_2,\ldots,x_{k+1}\rangle)$ per definition.

II: $f_{uptriangle}(A,B,\{C_1,\ldots,C_k],\langle x_2,\ldots,x_{k+1}\rangle) =$

$(\langle\rangle,C_1)\hat{}f_{uptriangle}(A,B,[C_2,\ldots,C_k],\langle x_2,\ldots,x_{k+1}\rangle)$ by induction of the length of C_1.

III: The lemma now follows by induction on k.

□

Lemma 5.3.3(Behaviour of f_{up}).

Let $1 \leq n \leq m$, $A = \{A_1,\ldots,A_n]$, $B = \{B_1,\ldots,B_m]\hat{}B'$, and for all $1 \leq i \leq n$ $|A_i| = |B_i|$, A_iB_i *the inproduct of* A_i *and* B_i.

Then $f_{up}(A,B,C) = ([B_1,\ldots,B_n],\langle\rangle)\hat{}f_{uptriangle}(\langle\rangle,B'',C,\langle A_1B_1,\ldots,A_nB_n\rangle)$ *for some* B".

Proof. I: $f_{up}(A,B,C) = (\langle \$\rangle,\langle\rangle)\hat{}f_{up'}(A,B,C,0,\langle\rangle)$ per definition.

II: $f_{up'}(\{A_1,\ldots,A_n],\{B_1,\ldots,B_m]\hat{}B',C,x,q) =$

$(B_1,\langle\rangle)\hat{}f_{up'}([A_2,\ldots,A_n],[B_2,\ldots,B_m]\hat{}B',x+A_1B_1,q)$ by induction on the length of A_1.

III: The lemma now follows by induction on n.

□

Lemma 5.3.4(Behaviour of f_{up0}).

Let $0 \leq n \leq m$, and for all $1 \leq i,j \leq n$ $|A_i| = |B_j|$.

Then $f_{up0}([A_1,\ldots,A_n],[B_1,\ldots,B_m]\hat{}B') = [R_1,\ldots,R_n]$

where $R_i = \langle A_iB_i,\ldots,A_nB_i\rangle$.

Proof. By induction on n.

Base: n=0 $f_{up0}([],B) = []$ per definition.

Step: n≧1 $f_{up0}([A_1,\ldots,A_n],[B_1,\ldots,B_m]\hat{}B')$

$= f_{up}(f_{dup}(\{A_1,\ldots,A_n\})\downarrow 1,\{B_1,\ldots,B_m\}\hat{}B',C)\downarrow 2$

$= f_{up}(\{A_1,\ldots,A_n\},\{B_1,\ldots,B_m\}\hat{}B',C)\downarrow 2$ (by Lemma 5.3.1.)

where $C = f_{up0}(f_{dup}(\{A_1,\ldots,A_n\})\downarrow 2,$

$f_{up}(f_{dup}(\{A_1,\ldots,A_n\})\downarrow 1,\{B_1,\ldots,B_m\}\hat{}B',C)\downarrow 1)$

$= f_{up0}([A_2,\ldots,A_n],f_{up}(\{A_1,\ldots,A_n\},\{B_1,\ldots,B_m\}\hat{}B',C)\downarrow 1)$
(by Lemma 5.3.1.)

$= f_{up0}([A_2,\ldots,A_n],(([B_1,\ldots,B_n],<>)\hat{}$

$f_{uptriangle}(<>,B'',C,<A_1B_1,\ldots,A_nB_n>))\downarrow 1)$
(by Lemma 5.3.3.)

$= f_{up0}([A_2,\ldots,A_n],[B_1,\ldots,B_n])$ (by Lemma 5.3.2.)

$= [R_2,\ldots,R_n]$ (induction hypothesis)

where $R_i = <A_iB_{i-1},\ldots,A_nB_{i-1}>$.

So $f_{up0}([A_1,\ldots,A_n],[B_1,\ldots,B_m]\hat{}B')$

$= f_{up}(\{A_1,\ldots,A_n\},\{B_1,\ldots,B_m\}\hat{}B',[R_2,\ldots,R_n])\downarrow 2$

$= f_{uptriangle}(<>,B'',[R_2,\ldots,R_n],<A_1B_1,\ldots,A_nB_n>)\downarrow 2$
(by Lemma 5.3.3.)

$= [R'_1,\ldots,R'_n]$ (by Lemma 5.3.2.)

where $R'_i = <A_iB_i>\hat{}R_{i+1} = <A_iB_i,\ldots,A_nB_i>$

$R'_n = <A_nB_n>$.

□

Lemmas 5.3.5. to 5.3.7., which are the low-counterparts of lemmas 5.3.2. to 5.3.4., are stated without proof.

Lemma 5.3.5(Behaviour of $f_{lowtriangle}$).

Let k≧0.

Then $f_{lowtriangle}(A,B,[C_2,\ldots,C_{k+1}],<x_1,\ldots,x_{k+1}>) =$

$(<>,[<x_1>,C_2\hat{}<x_2>,\ldots,C_{k+1}\hat{}<x_{k+1}>])$.

□

Lemma 5.3.6(Behaviour of f_{low}). *Same conditions as for lemma 5.3.3.*

$f_{low}(A,B,C) = ([B_1,\ldots,B_n],<>)\hat{}f_{lowtriangle}(<>,B'',C,<A_1B_1,\ldots,A_nB_n>)$.

□

Lemma 5.3.7(Behaviour of f_{low0}). *Same conditions as for lemma 5.3.4.*

$f_{low0}([A_1,\ldots,A_n],[B_1,\ldots,B_m]\hat{}B') = [R_1,\ldots,R_n]$

where $R_i = <A_iB_1,\ldots,A_iB_i>$.

□

Lemma 5.3.8(Behaviour of $f_{collect}$).

Let $k \geq 0$.

Then $f_{collect}(A,B,[C_1,\ldots,C_k],[D_2,\ldots,D_{k+1}],<x_1,\ldots,x_{k+1}>) =$

$(<>,<>,[R_1,\ldots,R_{k+1}])$

where $R_1 = <x_1>$ *(if* $k=0$*) or* $<x_1>\hat{}C_1$ *(otherwise)*

$R_i = D_i\hat{}<x_i>\hat{}C_i \quad 2 \leq i \leq k$

$R_{k+1} = D_{k+1}\hat{}<x_{k+1}>$.

Proof. I: $k=0$ $f_{collect}(A,B,[],[],<x_1>) = (<>,<>,[x_1]) = (<>,<>,[R_1])$.

II: $k=1$ $f_{collect}(A,B,[C_1],[D_2],<x_1,x_2>$

$= (<>,<>,<\$,x_1>)\hat{}f_{collectup}(A,B,\{C_1],\{D_2],<x_2>)$

$= (<>,<>,<\$,x_1>\hat{}C_1)\hat{}f_{collectup}(A,B,[],\{D_2],<x_2>)$

(by induction on the length of C_1)

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>)\hat{}f_{collectlow}(A,B,<>,\{D_2],<x_2>)$

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>\hat{}D_2)\hat{}f_{collectlow}(A,B,<>,[],<x_2>)$

(by induction on the length of D_2)

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>\hat{}D_2\hat{}<x_2,*>)$

$= (<>,<>,[R_1,R_2])$.

III: $k \geq 2$ $f_{collect}(A,B,[C_1,\ldots,C_k],[D_2,\ldots,D_{k+1}],<x_1,\ldots,x_{k+1}>)$

$= (<>,<>,<\$,x_1>)\hat{}f_{collectup}(A,B,\{C_1,\ldots,C_k],\{D_2,\ldots,D_{k+1}],$

$<x_2,\ldots,x_{k+1}>)$

$= (<>,<>,<\$,x_1>\hat{}C_1)\hat{}f_{collectup}(A,B,[C_2,\ldots,C_k],\{D_2,\ldots,D_{k+1}],$

$<x_2,\ldots,x_{k+1}>)$

(by induction on the length of C_1)

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>)\hat{}f_{collectlow}(A,B,\{C_2,\ldots,C_k],$

$\{D_2,\ldots,D_{k+1}],<x_2,\ldots,x_{k+1}>)$

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>\hat{}D_2\hat{}<x_2>)\hat{}f_{collectlow}(A,B,\{C_2,\ldots,C_k],$

$[D_3,\ldots,D_{k+1}],<x_3,\ldots,x_{k+1}>)$

(by induction on the length of D_2)

$= (<>,<>,<\$,x_1>\hat{}C_1\hat{}<\$>\hat{}D_2)\hat{}f_{collectup}(A,B,[C_2,\ldots,C_k],$

$[D_3,\ldots,D_{k+1}],<x_2,\ldots,x_{k+1}>).$

IV: The lemma follows by induction on k. (Base: k=1)

□

Lemma 5.3.9(Behaviour of f_{centre}). *Same conditions as for lemma 5.3.4.*

$f_{centre}(\{A_1,\ldots,A_n],\{B_1,\ldots,B_n],C,D) =$

$([A_1,\ldots,A_n],[B_1,\ldots,B_n],<>)\hat{}f_{collect}(<>,<>,C,D,<A_1B_1,\ldots,A_nB_n>).$

Proof. *Same as lemma 5.3.3.*

□

Theorem 5.3.10(Behaviour of f_{madm}).

Let $n\geq 0$ and for all $1\leq i,j\leq n$ $|A_i| = |B_j|$.

Then $f_{madm}([A_1,\ldots,A_n],[B_1,\ldots,B_n]) = [R_1,\ldots,R_n]$

where $R_i = <A_iB_1,\ldots,A_iB_n>$.

Proof. n=0 $f_{madm}([],[]) = []$ per definition

$n\geq 1$ $f_{madm}([A_1,\ldots,A_n],[B_1,\ldots,B_n])$

$= f_{centre}(f_{dup}(\{A_1,\ldots,A_n])\downarrow 1,f_{dup}(\{B_1,\ldots,B_n])\downarrow 1,C,D)\downarrow 3$

$= f_{centre}(\{A_1,\ldots,A_n],\{B_1,\ldots,B_n],C,D)\downarrow 3$

where $C = f_{up0}(f_{dup}(\{B_1,\ldots,B_n])\downarrow 2,f_{centre}(f_{dup}(\{A_1,\ldots,A_n])\downarrow 1,$

$f_{dup}(\{B_1,\ldots,B_n]\downarrow 1,C,D)\downarrow 1)$

$= f_{up0}([B_2,\ldots,B_n],[A_1,\ldots,A_n])\hat{}A'$ (for some A')

(by Lemmas 5.3.1., 5.3.9.)

$= [T_1,\ldots,T_{n-1}]$ (by Lemma 5.3.4.)

where $T_i = <A_iB_{i+1},\ldots,A_iB_n>$

and $D = [S_2,\ldots,S_n]$

where $S_i = \langle A_iB_i,\ldots,A_iB_{i-1}\rangle$ (same reasoning as for C).

Combining this we get (by Lemma 5.3.9.)

$f_{centre}(\{A_1,\ldots,A_n],\{B_1,\ldots,B_n],C,D)\downarrow 3$

$= f_{centre}(\{A_1,\ldots,A_n],\{B_1,\ldots,B_n],[T_1,\ldots,T_{n-1}],[S_2,\ldots,S_n])\downarrow 3$

$= f_{collect}(\langle\rangle,\langle\rangle,[T_1,\ldots,T_{n-1}],[S_2,\ldots,S_n],\langle A_1B_1,\ldots,A_nB_n\rangle)\downarrow 3$

We now apply lemma 5.3.8.:

I: $n=1$ $f_{collect}(\langle\rangle,\langle\rangle,[],[],\langle A_1B_1\rangle)\downarrow 3 = [A_1B_1]$

II: $n\geq 2$ $f_{collect}(\langle\rangle,\langle\rangle,[T_1,\ldots,T_{n-1}],[S_2,\ldots,S_n],\langle A_1B_1,\ldots,A_nB_n\rangle)$

$= [R_1,\ldots,R_n]$

where $R_1 = \langle A_1B_1\rangle\hat{}T_1 = \langle A_1B_1\rangle\hat{}\langle A_1B_2,\ldots,A_1B_n\rangle$

$R_i = S_i\hat{}\langle A_iB_i\rangle\hat{}T_i = \langle A_iB_1,\ldots,A_iB_{i-1}\rangle\hat{}\langle A_iB_i\rangle\hat{}$

$\langle A_iB_{i+1},\ldots,A_iB_n\rangle$

$R_n = S_n\hat{}\langle A_nB_n\rangle = \langle A_nB_1,\ldots,A_nB_{n-1}\rangle\hat{}A_nB_n$.

□

5.4. CORRECTNESS OF DIVCONQ

We prove the correctness of Divconq (figure 4.2.3.7.) independently of the precise specification of the primitive functions solve-seq, size, combine, and split. These primitives must, however, have certain properties which we discuss first.

The sequential program solve-seq takes a problem p and yields a result r, where p is an element of the problem domain P and r an element of the result domain R.

$$f_{\text{solve-seq}} : P \rightarrow R$$

We will show that Divconq behaves just like solve-seq:

$$\forall p \in P : f_{\text{divconq}}(p) = f_{\text{solve-seq}}(p)$$

The primitive function size measures the size of a problem. The size of a problem is a positive integer.

$$f_{\text{size}} : P \rightarrow \mathbb{N}$$

If the size of a problem is one, we call it a simple problem. The primitive function split takes a non-simple problem and yields two problems p1 and p2. We will use two functions to describe split.

$$f_{\text{split1}} : P^{NS} \rightarrow P$$

$$f_{\text{split2}} : P^{NS} \rightarrow P$$

$$\text{where } P^{NS} = P \setminus \{p \mid f_{\text{size}}(p) = 1\}$$

$$\text{such that } f_{\text{size}}(f_{\text{split1}}(p)) = \lfloor f_{\text{size}}(p)/2 \rfloor$$

$$f_{\text{size}}(f_{\text{split2}}(p)) = \lceil f_{\text{size}}(p)/2 \rceil$$

The primitive function combine takes two results and yields one result.

$$f_{\text{combine}} : R \times R \rightarrow R$$

$$\text{such that } \forall p \in P^{NS} : f_{\text{combine}}(f_{\text{solve-seq}}(f_{\text{split1}}(p)), f_{\text{solve-seq}}(f_{\text{split2}}(p))) = f_{\text{solve-seq}}(p)$$

In the sequel the / operator will perform a truncation towards zero for positive operands, just as in C. We have introduced one extra primitive function twolog. The only property we demand of twolog is:

$$\forall p \in P : 1 \leq f_{size}(p)/g(f_{size}(p)) \leq f_{size}(p)$$

where g(n) denotes the meaning of twolog.

We will assume no other properties of the primitives than the ones stated above. We will now derive the functions describing Divconq. The initial network is shown in figure 5.4.1.

$$f_{divconq}(prb) = f_{root}(prb,Y)\downarrow 1$$

$$\text{where } Y = f_{leaf}(f_{root}(prb,Y)\downarrow 2)$$

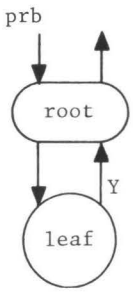


Figure 5.4.1. Initial network of Divconq.

The process root determines whether it must split the problem. If not, it just solves the problem sequentially. This is the only case where a leaf process is needlessly created, and could have been prevented by making the main process Divconq check for it. If the problem has to be split, root first sends down a measure of the row of problems which will follow, and then goes into a "splitloop" followed by a matching "combineloop".

$$f_{root}(prb,subress) =$$

$$[x=1 \rightarrow (\langle f_{solve\text{-}seq}(F(prb)),EOF\rangle,\langle EOF\rangle)$$

$$, (\langle\rangle,\langle x/2\rangle)\hat{}f_{splitloop}(R(prb),subress,x,x,F(prb))]$$

$$\text{where } x = f_{size}(F(prb))/g(f_{size}(F(prb)))$$

$$f_{splitloop}(prb,subress,c,n,p) =$$

$$[c/2>0 \rightarrow (<>,<f_{split2}(p)>)\hat{}f_{splitloop}(prb,subress,c/2,n,f_{split1}(p))$$

$$, f_{combineloop}(prb,subress,n,f_{solve\text{-}seq}(p))]$$

$$f_{combineloop}(prb,subress,c,r) =$$

$$[c/2>0 \rightarrow f_{combineloop}(prb,R(subress),c/2,f_{combine}(r,F(subress)))$$

$$, (<r,EOF>,<EOF>)]$$

A leaf process reads the measure of the row of problems it will receive. If it will receive one problem it will solve that problem sequentially, otherwise it will expand as shown in figure 5.4.2.

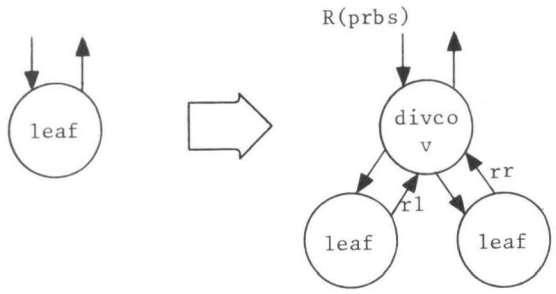


Figure 5.4.2. Expansion of leaf.

$$f_{leaf}(prbs) = [F(prbs) = EOF \rightarrow <EOF>,$$

$$[F(prbs) = 1 \rightarrow <f_{solve\text{-}seq}(F(R(prbs))),EOF>$$

$$, f_{divco}(R(prbs),rl,rr,F(prbs))\downarrow 1]]$$

$$\text{where } rl = f_{leaf}(f_{divco}(R(prbs),rl,rr,F(prbs))\downarrow 2)$$

$$rr = f_{leaf}(f_{divco}(R(prbs),rl,rr,F(prbs))\downarrow 3)$$

A divco process splits and divides all problems it receives except the last one. The last problem is solved sequentially and the result is sent up. Then, the results of the problems sent down are combined and sent up.

$f_{divco}(prbs,rl,rr,n) = (<>,<n/2>,<n/2>)\hat{}f_{doublesplit}(prbs,rl,rr,n,n)$

$f_{doublesplit}(prbs,rl,rr,c,n) =$

$[c/2>0 \rightarrow (<>,<f_{split1}(F(prbs))>,<f_{split2}(F(prbs))>)\hat{}$

$f_{doublesplit}(R(prbs),rl,rr,c/2,n)$

$, (<f_{solve\text{-}seq}(F(prbs))>,<>,<>)\hat{}$

$f_{doublecombine}(R(prbs),rl,rr,n)]$

$f_{doublecombine}(prbs,rl,rr,c) =$

$[c/2>0 \rightarrow (<f_{combine}(F(rl),F(rr))>,<>,<>)\hat{}$

$f_{doublecombine}(prbs,R(rl),R(rr),c/2)$

$, (<EOF>,<EOF>,<EOF>)]$

The rows of problems and results travelling over the channels have certain well-formedness properties which are defined as follows:

Definition 5.4.1(Well-formed row of problems).

Let $c \in \mathbb{N}$ and $p_k,\ldots,p_0 \in P$.

Then (1) $<0>$ is an (empty but) *well-formed row of problems*, written $<0>$ *WFP*

(2) $<c,p_k,\ldots,p_0>$ is a *well-formed row of problems*, written $<c,p_k,\ldots,p_0>$ *WFP*

iff: (i) $2^k \leq c < 2^{k+1}$ $(k \geq 0)$

(ii) $f_{size}(p_i) \geq 2^i$

□

Definition 5.4.2(Well-formed row of results).

Let $c \in \mathbb{N}$, and $r_k,\ldots,r_1 \in R$.

Then $<c,r_1,\ldots,r_k>$ is a *well-formed row of results*, written $<c,r_1,\ldots,r_k>$ *WFP*,

iff $2^k \leq c < 2^{k+1}$.

□

We also define the transitive closure of $f_{combine}$.

Definition 5.4.3. $f^*_{combine} : R \times R^* \to R$

$f^*_{combine}(r,<>) = r$

$f^*_{combine}(r_0,r_1\hat{}X) = f^*_{combine}(f_{combine}(r_0,r_1),X)$

□

Clearly $f^*_{combine}(r_0,X\hat{}r) = f_{combine}(f^*_{combine}(r_0,X),r)$.

Lemma 5.4.1(Behaviour of $f_{combineloop}$).

Let $<c,r_1,\ldots,r_k>$ WFR.

Then $f_{combineloop}(prb,<r_1,\ldots,r_k>\hat{}X,c,r) =$

$(<f^*_{combine}(r,<r_1,\ldots,r_k>),EOF>,<EOF>)$.

Proof. By induction on c.

Base: c=1

$f_{combineloop}(prb,X,1,r) = (<r,EOF>,<EOF>) =$

$(<f^*_{combine}(r,<>),EOF>,<EOF>)$.

Step: c>1

$f_{combineloop}(prb,<r_1,\ldots,r_k>\hat{}X,c,r) =$

$f_{combineloop}(prb,<r_2,\ldots,r_k>\hat{}X,c/2,f_{combine}(r,r_1))$

c>1 and $(c,r_1,\ldots,r_k)$ WFR → $(c/2,r_2,\ldots,r_k)$ WFR

So we may apply the induction hypothesis:

$f_{combineloop}(prb,<r_2,\ldots,r_k>\hat{}X,c/2,f_{combine}(r,r_1)) =$

$(<f^*_{combine}(f_{combine}(r,r_1),<r_2,\ldots,r_k>),EOF>,<EOF>) =$

$(<f^*_{combine}(r,<r_1,\ldots,r_k>),EOF>,<EOF>)$.

□

Lemma 5.4.2(Behaviour of $f_{splitloop}$).

Let $f_{size}(p)\geq c$, $c\geq 1$, $n\geq c$.

Then $f_{splitloop}(prb,subress,c,n,p) =$

$(<>,<p_{k-1},\ldots,p_0>)\hat{}f_{combineloop}(prb,subress,n,f_{solve\text{-}seq}(\overline{p}_0))$

where (1) $<c/2,p_{k-1},\ldots,p_0>$ WFP

(2) $f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),$

$<f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-1})>) = f_{solve\text{-}seq}(p)$.

Proof. By induction on c.

Base: c=1

$$f_{splitloop}(prb,subress,1,n,p)$$

$$= f_{combineloop}(prb,subress,n,f_{solve\text{-}seq}(p))$$

$\overline{p}_0=p,\langle p_{k-1},\ldots,p_0\rangle = \langle\rangle$ and (1) and (2) are satisfied.

Step: $c\geq 2$

$$f_{splitloop}(prb,subress,c,n,p)$$

$$= (\langle\rangle,\langle f_{split2}(p)\rangle)^\frown f_{splitloop}(prb,subress,c/2,n,f_{split1}(p))$$

Now $f_{size}(p)\geq c \rightarrow f_{size}(f_{split1}(p))\geq c/2$.

So we may apply the induction hypothesis:

$$f_{splitloop}(prb,subress,c,n,p)$$

$$= (\langle\rangle,\langle f_{split2}(p)\rangle)^\frown(\langle\rangle,\langle p_{k-2},\ldots,p_0\rangle)^\frown f_{combineloop}(prb,subress,n,f_{solve\text{-}seq}(\overline{p}_0))$$

$$= (\langle\rangle,\langle f_{split2}(p),p_{k-2},\ldots,p_0\rangle)^\frown f_{combineloop}(prb,subress,n,f_{solve\text{-}seq}(\overline{p}_0))$$

where (1) $\langle c/4,p_{k-2},\ldots,p_0\rangle$ WFP

(2) $f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),\langle f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-2})\rangle) = f_{solve\text{-}seq}(f_{split1}(p))$.

Now we must check (1) and (2) for $f_{splitloop}(prb,subress,c,n,p)$:

(1) $f_{size}(p)\geq c \rightarrow f_{size}(f_{split2}(p))\geq c/2$

and therefore $\langle c/2,f_{split2}(p),p_{k-2},\ldots,p_0\rangle$ WFP,

(2) $f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),\langle f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-2}),f_{solve\text{-}seq}(f_{split2}(p))\rangle)$

$$= f_{combine}(f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),\langle f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-2})\rangle),f_{solve\text{-}seq}(f_{split2}(p)))$$

$$= f_{combine}(f_{solve\text{-}seq}(f_{split1}(p)),f_{solve\text{-}seq}(f_{split2}(p)))$$

$$= f_{solve\text{-}seq}(p).$$

□

Lemma 5.4.3(Behaviour of $f_{doublecombine}$).

Let $\langle c,r'_1,\ldots,r'_k\rangle$ WFR, $\langle c,r''_1,\ldots,r''_k\rangle$ WFR.

Then $f_{doublecombine}(X,\langle r'_1,\ldots,r'_k\rangle\hat{}Y,\langle r''_1,\ldots,r''_k\rangle\hat{}Z,c) =$

$(\langle r_1,\ldots,r_k,EOF\rangle,\langle EOF\rangle,\langle EOF\rangle)$

where $r_i = f_{combine}(r'_i,r''_i)$.

Proof. By induction on c.

Base: c=1

$f_{doublecombine}(X,Y,Z,1) = (\langle EOF\rangle,\langle EOF\rangle,\langle EOF\rangle)$

Step: $c\geq 2$

$f_{doublecombine}(X,\langle r'_1,\ldots,r'_k\rangle\hat{}Y,\langle r''_1,\ldots,r''_k\rangle\hat{}Z,c)$

$= (\langle f_{combine}(r'_1,r''_1)\rangle,\langle\rangle,\langle\rangle)\hat{}$

$f_{doublecombine}(X,\langle r'_2,\ldots,r'_k\rangle\hat{}Y,\langle r''_2,\ldots,r''_k\rangle\hat{}Z,c/2)$

Now $\langle c/2,r'_2,\ldots,r'_k\rangle$ WFR and $\langle c/2,r''_2,\ldots,r''_k\rangle$ WFR

So we may apply the induction hypothesis:

$f_{doublecombine}(X,\langle r'_1,\ldots,r'_k\rangle\hat{}Y,\langle r''_1,\ldots,r''_k\rangle\hat{}Z,c)$

$= (\langle r_1\rangle,\langle\rangle,\langle\rangle)\hat{}(\langle r_2,\ldots,r_k,EOF\rangle,\langle EOF\rangle,\langle EOF\rangle)$.

□

Lemma 5.4.4(behaviour of $f_{doublesplit}$).

Let $\langle c,p_k,\ldots,p_0\rangle$ WFP, $c\geq 1$, $n\geq c$.

Then $f_{doublesplit}(\langle p_k,\ldots,p_0\rangle\hat{}X,rl,rr,c,n)$

$= (\langle f_{solve\text{-}seq}(p_0)\rangle,\langle p'_{k-1},\ldots,p'_0\rangle,\langle p''_{k-1},\ldots,p''_0\rangle)\hat{}$

$f_{doublecombine}(X,rl,rr,n)$

where $p'_i = f_{split1}(p_{i+1})$ *and* $\langle c/2,p'_{k-1},\ldots,p'_0\rangle$ WFP

$p''_i = f_{split2}(p_{i+1})$ *and* $\langle c/2,p''_{k-1},\ldots,p''_0\rangle$ WFP.

Proof. By induction on c.

Base: c=1

$f_{doublesplit}(\langle p_0\rangle\hat{}X,rl,rr,1,n)$

$= (\langle f_{solve\text{-}seq}(p_0)\rangle,\langle\rangle,\langle\rangle)\hat{}f_{doublecombine}(X,rl,rr,n)$

Step: $c \geq 2$

$$f_{doublesplit}(\langle p_k,\ldots,p_0\rangle \hat{} X, rl, rr, c, n)$$

$$= (\langle\rangle, \langle f_{split1}(p_k)\rangle, \langle f_{split2}(p_k)\rangle) \hat{}$$

$$f_{doublesplit}(\langle p_{k-1},\ldots,p_0\rangle \hat{} X, rl, rr, c/2, n)$$

Because $\langle c/2, p_{k-1},\ldots,p_0\rangle$ WFP, the induction hypothesis applies:

$$f_{doublesplit}(\langle p_k,\ldots,p_0\rangle \hat{} X, rl, rr, c, n)$$

$$= (\langle\rangle, \langle p'_{k-1}\rangle, \langle p''_{k-1}\rangle) \hat{} (\langle f_{solve\text{-}seq}(p_0)\rangle, \langle p'_{k-2},\ldots,p'_0\rangle, \langle p''_{k-2},\ldots,p''_0\rangle) \hat{}$$

$$f_{doublecombine}(X, rl, rr, n)$$

$$= (\langle f_{solve\text{-}seq}(p_0)\rangle, \langle p'_{k-1},\ldots,p'_0\rangle, \langle p''_{k-1},\ldots,p''_0\rangle) \hat{}$$

$$f_{doublecombine}(X, rl, rr, n)$$

where $p'_i = f_{split1}(p_{i+1})$

$p''_i = f_{split2}(p_{i+1})$

and, because $f_{size}(p'_{k-1}) = f_{size}(f_{split1}(p_k)) \geq c/2$

$f_{size}(p''_{k-1}) = f_{size}(f_{split2}(p_k)) \geq c/2$

(and induction hypothesis) we have:

$\langle c/2, p'_{k-1},\ldots,p'_0\rangle$ WFP

$\langle c/2, p''_{k-1},\ldots,p''_0\rangle$ WFP.

□

Lemma 5.4.5(Behaviour of f_{leaf}).

Let $\langle c, p_k,\ldots,p_0\rangle$ WFP.

Then $f_{leaf}(\langle c, p_k,\ldots,p_0\rangle \hat{} X) = \langle r_0,\ldots,r_k, EOF\rangle$

where $r_i = f_{solve\text{-}seq}(p_i)$.

Proof. By induction on c.

Base: c=1

$$f_{leaf}(\langle 1, p_0\rangle \hat{} X) = \langle f_{solve\text{-}seq}(p_0), EOF\rangle.$$

Step: $c \geq 2$

$$f_{leaf}(<c,p_k,\ldots,p_0>\hat{}X) = f_{divco}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c)\downarrow 1.$$

where $rl = f_{leaf}(f_{divco}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c)\downarrow 2)$

$rr = f_{leaf}(f_{divco}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c)\downarrow 3)$.

$f_{divco}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c)$

$= (<>,<c/2>,<c/2>)\hat{}f_{doublesplit}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c,c)$

$= (<f_{solve\text{-}seq}(p_0)>,<c/2,p'_{k-1},\ldots,p'_0>,<c/2,p''_{k-1},\ldots,p''_0>)\hat{}$

$f_{doublecombine}(X,rl,rr,c)$ (by Lemma 5.4.4.)

where $p'_i = f_{split1}(p_{i+1})$ and $<c/2,p'_{k-1},\ldots,p'_0>$ WFP

$p''_i = f_{split2}(p_{i+1})$ and $<c/2,p''_{k-1},\ldots,p''_0>$ WFP

$rl = f_{leaf}(<c/2,p'_{k-1},\ldots,p'_0>\hat{}X') = <r'_0,\ldots,r'_{k-1},EOF>$

(because $<c/2,p'_{k-1},\ldots,p'_0>$ WFP so the induction hypothesis applies)

where $r'_i = f_{solve\text{-}seq}(p'_i) = f_{solve\text{-}seq}(f_{split1}(p_{i+1}))$

$rr = <r''_0,\ldots,r''_{k-1},EOF>$

where $r''_i = f_{solve\text{-}seq}(p''_i) = f_{solve\text{-}seq}(f_{split2}(p_{i+1}))$.

$f_{divco}(<p_k,\ldots,p_0>\hat{}X,rl,rr,c)\downarrow 1$

$= <f_{solve\text{-}seq}(p_0)>\hat{}f_{doublecombine}(X,<r'_0,\ldots,r'_{k-1},EOF>,$

$<r''_0,\ldots,r''_{k-1},EOF>,c)\downarrow 1$

($<c,p_k,\ldots,p_0>$ WFP implies $<c,r'_0,\ldots,r'_{k-1}>$ WFR → Lemma 5.4.3.)

$= <f_{solve\text{-}seq}(p_0)>\hat{}<r_0,\ldots,r_{k-1},EOF>$

where $r_i = f_{combine}(r'_i,r''_i)$

$= f_{combine}(f_{solve\text{-}seq}(f_{split1}(p_{i+1})),$

$f_{solve\text{-}seq}(f_{split2}(p_{i+1})))$

$= <f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_k),EOF>.$

□

Theorem 5.4.6. $f_{divconq}(<p>\hat{}X) = <f_{solve\text{-}seq}(p),EOF>$.

Proof. $f_{divconq}(<p>\hat{}X) = f_{root}(<p>\hat{}X,Y)\downarrow 1$

where $Y = f_{leaf}(f_{root}(<p>\hat{}X,Y)\downarrow 2)$.

There are two cases:

$x = f_{size}(p)/g(f_{size}(p)) = 1$:

$f_{root}(<p>\hat{}X,Y) = (<f_{solve\text{-}seq}(p),EOF>,<EOF>)$.

$x>1$:

$f_{root}(<p>\hat{}X,Y) = (<>,<x/2>)\hat{}f_{splitloop}(X,Y,x,x,p)$

(Lemma 5.4.2. applies because $f_{size}(p)\geq x$)

$= (<>,<x/2>)\hat{}(<>,<p_{k-1},..,p_0>)\hat{}f_{combineloop}(X,Y,x,f_{solve\text{-}seq}(\overline{p}_0)) =$

$= (<>,<x/2,p_{k-1},\ldots,p_0>\hat{}f_{combineloop}(X,Y,x,f_{solve\text{-}seq}(\overline{p}_0))$.

where (1) $<x/2,p_{k-1},\ldots,p_0>$ WFP

(2) $f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),$

$<f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-1})>)$

$= f_{solve\text{-}seq}(p)$.

$f_{root}(<p>\hat{}X,Y)\downarrow 2 = <x/2,p_{k-1},\ldots,p_0>\hat{}X'$

We therefore have that

$Y = f_{leaf}(<x/2,p_{k-1},\ldots,p_0>\hat{}X')$

$= <r_0,\ldots,r_{k-1},EOF>$ (by Lemma 5.4.5.)

where $r_i = f_{solve\text{-}seq}(p_i)$.

$f_{root}(<p>\hat{}X,Y)\downarrow 1$

$= f_{combineloop}(X,<r_0,\ldots,r_{k-1},EOF>,x,f_{solve\text{-}seq}(\overline{p}_0))\downarrow 1$

$= <f^*_{combine}(f_{solve\text{-}seq}(\overline{p}_0),<f_{solve\text{-}seq}(p_0),\ldots,f_{solve\text{-}seq}(p_{k-1})>),EOF>$

(by Lemma 5.4.1.)

$= <f_{solve\text{-}seq}(p),EOF>$.

□

5.5. REMARKS

The above sections show that realistic DNP programs can be proved correct using Kahn's two step method of translating programs into sets of equations and solving these equations. The proofs are long (about the size of the programs) because we have to deal with many details. The proofs may be stated in a more direct way using a Hoare style system for the language. At this moment work is being done to construct such Hoare style proof rules and axioms [21].

Another drawback of the semantics used here is that we can only prove properties of the complete histories travelling over the channels (because these are the solutions of the equations). We cannot make statements about the relative ordering of say, reads and writes in various processes which might be the very purpose of a certain program (such as the implementation of a protocol). In a Hoare style system one may prove properties of this kind.

REFERENCES

[1] ACKERMAN, W.B., DENNIS, J.B., *VAL - A Value oriented Algorithmic language. Preliminary reference manual*, MIT/LCS/TR-218, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1979.

[2] ADAMS, D.A., *A Computational Model with Data Flow Sequencing*, TR/CS-117, School of Humanities and Sciences, Stanford Univ., Stanford, California, 1968.

[3] ALLAN, S.J., OLDEHOEFT, A.E., *A flow analysis Procedure for the Translation of High-level languages to a Data Flow Language*, IEEE Transactions on Computers, 29,9 (1980), pp. 826-831.

[4] ALLEN, A.O., *Probability, Statistics and Queuing Theory. With Computer Science Applications*, Academic Press, New York, 1978.

[5] ARVIND, GOSTELOW, K.P., *Some Relationships between asynchronous interpreters of a dataflow language*, in NEUHOLD, E.J. (ed.), *Formal descriptions of Programming Concepts*, North Holland Publishing Company, New York, 1977, pp. 95-119.

[6] ARVIND, IANUCCI, R.A., *A Critique of Multiprocessing von Neumann Style*, MIT/LCS/TM-226, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1983.

[7] ASHCROFT, E.A., WADGE, W.W., *LUCID, a Nonprocedural Language with Iteration*, CACM, 20,7 (1977), pp. 519-526.

[8] BACKUS, J., *Can Programming be liberated from the von Neumann Style? A functional Style and its Algebra of Programs*, CACM, 21,8 (1978), pp. 631-641.

[9] BATCHER, K.E., *Sorting networks and their applications*, Proc. AFIPS 1968 SJCC, 32, AFIPS Press, Montvale, N.J., 1968, pp. 307-314.

[10] BÖHM, A.P.W., DE BRUIN, A., *Dynamic Networks of Parallel Processes*, IW 192/82, Dept. of Comp. Sci., Mathematisch Centrum, Amsterdam, 1982.

[11] BOUSSINOT, F., *Proposition de Sémantique dénotationelle pour des réseaux de processus avec operateur de mélange equitable*, Report 2487, Thomson CSF, Corbeville, France, 1980.

[12] BOWEN, D.L., *Implementation of Data Structures in a Dataflow Computer*, Ph.D. Thesis, Dept. of Comp. Sci., Victoria University of Manchester,1981.

[13] BROCK, J.D., *Operational Semantics of a Data Flow Language*, MIT/LCS/TM-120, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1978.

[14] DE BRUIN, A., BÖHM, A.P.W., *The denotational semantics of dynamic networks of processes*, RUU-CS-82-13, Dept. of Comp. Sci., Univ. of Utrecht, 1982.

[15] BURKS, A.W., GOLDSTINE, H.H., VON NEUMANN, J., *Preliminary discussion of the logical design of an electronic computing instrument*, U.S. Army Ordnance Department Report, 1946.

[16] CATTO, A.J., GURD, J., *Nondeterministic Dataflow Graphs*, Proc. 8th IFIP World Computer Congress. IFIP80, North Holland Publishing Company, Amsterdam, 1980, pp. 251-256.

[17] CHAMBERLIN, D.D., *The 'Single-Assignment' Approach to Parallel Processing*, AFIPS Conf. Proc. 39, 1971 FJCC, 1971, pp. 263-269.

[18] COOK, S.A., *The complexity of theorem proving procedures*, Proc. 3rd. Ann. ACM Symposium on the theory of Computing, ACM, New York, 1971, pp. 151-158.

[19] DAVIS, A.L., *A data flow evaluation system based on the concept of recursive locality*, AFIPS Proc. of the National Computer Conf., New York, 1979, pp. 1079-1086.

[20] DAVIS, M., *Computability and unsolvability*, McGraw-Hill, New York, 1958.

[21] DE ROEVER, W.P., DE BRUIN, A., ZWIERS, J., *A sound proof system for dynamic networks of processes*, in CLARKE, E., KOZEN, D.(eds.), *Proc. 2nd Workshop on Logics of Programming*, (to appear in LNCS), 1983.

[22] DENNIS, J.B., *First Version of a Data Flow Procedure Language*, LNCS 19, Springer-Verlag, Berlin, 1974, pp. 362-376.

[23] DENNIS, J.B., MISUNAS, D.P., *A Preliminary Architecture for a Basic Data Flow Architecture*, The Second Ann. Symp. on Comp. Architecture: Conference Proceedings, IEEE, 1975, pp. 126-132.

[24] DENNIS, J.B., *Data Flow Supercomputers*, IEEE Computer, Nov. 1980, pp. 48-56.

[25] EARLEY, J., STURGIS, H., *A Formalism for translator Interactions*, CACM, 13,10 (1970), pp. 607-617.

[26] FAUSTINI, A.A., *An Operational Semantics for Pure Dataflow*, in NIELSEN, M., SCHMIDT, E.M. (eds.), *Autamata, Languages and Programming*, Ninth Colloquium (Aarhus, Denmark), LNCS140, Springer-Verlag, Berlin, 1982, pp.

[27] FLORIJN, G., ROLF, G., *PGEN - A General Purpose Parser Generator*, IW 157/81, Dept. of Comp. Sci., Mathematisch Centrum, Amsterdam, 1981.

[28] FLYNN, M.J., *Very High-Speed Computing Systems*, Proc. IEEE, 54,12 (1966), pp. 1901-1909.

[29] FOSSEEN, J.B., *Representation of Algorithms bij maximally Parallel Schemata*, M.Sc. Thesis, Dept. of Electr. Eng., MIT, Cambridge, Mass, 1972.

[30] FRIEDMAN, D.P., WISE D.S., *Cons should not evaluate its arguments*, Automata languages and programming, Edinburgh Univ. Press, Edinburgh, Scotland, 1976, pp. 257-284.

[31] GAJSKI, D.P., PADUA, D.A., KUCK, D.J., KUHN, R.H., *A Second Opinion on Data Flow Machines and Programming*, IEEE Computer, febr. 1982, pp. 58-69.

[32] GAREY, M.R., JOHNSON, D.S., *Computers and Intractability - A guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[33] GHEZZI, C., DELLA VIGNA, P.L., *Context free graph grammars*, Information and Control, 37 (1978), pp. 207-233.

[34] GLAUERT, J.R.W., *A Single-Assignment Language for Dataflow Computing*, M.Sc. Dissertation, Dept. of Comp. Sci., Victoria University of Manchester, 1978.

[35] GURD, J., WATSON, I., *A Prototype data flow computer with token labelling*, AFIPS proc. of the National Computer Conf., New York, 1979, pp. 623-682.

[36] GURD, J., WATSON, I., *Data Driven system for high speed parallel computing*, part 1 and 2, Computer Design, June/July (1980), pp. 91-99/ pp. 97-105.

[37] HANKIN, C.L., GLASER, H.W., *The Data Flow Programming Language CAJOLE*, SICPLAN Notices, 16,7 (1981), pp. 35-44.

[38] HANSEN, P.B., *Operating Systems Principles*, Prentice Hall Inc., Englewood Cliffs, N.J., 1973.

[39] HOARE, C.A.R., *Communicating Sequential Processes*, CACM, 21,8 (1978), pp. 666-677.

[40] HOCKNEY, R.W., and JESSHOPE, C.R., *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981.

[41] HOEY, D., LEISERSON. C.E., *A layout for the Perfect Shuffle Exchange Network*, CMU-CS-80-139, Dept. of Comp. Sci., CMU, Pittsburg, 1980.

[42] HOPCROFT, J.E., ULLMAN, J.D., *Introduction to Automata theory, Languages, and Computation*, Addison-Wesley Publishing Company, Reading, Mass, 1979.

[43] HOSSFELD, F., *Parallelprozessoren und Algorithmenstruktur*, Bericht Jül-Spez. 87, Zentralinstitut f. angewandte Mathematik, Kernforschungsanlage Jülich GmbH, Jülich, 1980.

[44] JAFFE, J.M., *The equivalence of r.e. program schemes and dataflow schemes*, Journal of Computer and System science, 21 (1980), pp. 92-109.

[45] JENSEN, J.C., *Basic Program Representation in the Texas Instruments Data Test Bed Compiler*, Unpublished Memo, Texas Instruments Inc., 1980.

[46] KAHN, G., *The Semantics of a simple language for parallel programming*, IFIP74, North Holland Publishing Company, Amsterdam, 1974, pp. 471-475.

[47] KARP, R.M., and MILLER, R.E., *Properties of a model for parallel computations: determinacy, termination, queuing*, SIAM Journal of applied Mathematics, 14,6 (1966), pp. 1390-1411.

[48] KELLER, R.M., JAYARAMA, B., ROSE, D., Lindstrom, G., *FGL Programmers Guide*, Technical Memo 1, Dept. of Comp. Sci., Univ. of Utah, Salt Lake City, 1980.

[49] KERNIGHAN, B.W., RITCHIE, D.M., *The C programming language*, Prentice Hall Software Series, Englewood Cliffs, N.J., 1978.

[50] KLEENE, S.C., *General recursive functions of natural numbers*, Mathematische Annalen, 112 (1936), pp. 727-742.

[51] KLINT, P., *Summer Reference Manual*, Dept. of Comp. Sci., Mathematisch Centrum, Amsterdam, 1981.

[52] KNUTH, D.E., *The art of Computer programming*, Vol 1/*Fundamental Algorithms*, Addison Wesley Publishing Company, Reading, Mass, 1973.

[53] KNUTH, D.E., *Semantics of context free languages*, Mathematical Systems Theory, 2,2 (1968), pp. 127-145.

[54] KOSTER, C.H.A., *Affix Grammars*, in PECK, J.E.L.(ed.), *Algol 68 Implementation*, North Holland Publ. Co., Amsterdam, 1971, pp. 95-106.

[55] KRAMER, M.R., VAN LEEUWEN, J., *Systolic Computation and VLSI*, RUU-CS-82-9, Dept. of Comp. Sci., University of Utrecht, 1982.

[56] KUCK, D.J., *A Survey of Parallel Machine Organisation and Programming*, ACM Computing Surveys, 9,1 (1977), pp. 29-59.

[57] KUNG, H.T., LEISERSON, C.E., *Systolic Arrays(for VLSI)*, in DUFF, I.S., STEWART, I.S.(eds.), *Sparse Matrix Proc. 1978*, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

[58] KUNG, S.Y., ARUN, K.S., GAL-EZER, R.J., BHASKAR RAO, D.V., *Wave front Array Processor: Language, Architecture and Applications*, IEEE Transactions on Computers, C31,11 (1982), pp. 1054-1066.

[59] LEISERSON, C.E., *Systolic Priority Queues*, CMU-CS-79-115, Dept. of Comp. Sci., CMU, Pittsburg, 1979.

[60] LISKOV, B.H., et.al., *CLU Reference Manual*, Memo 161, Comp. Structures Group, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1978.

[61] MCILROY, M.D., *Coroutines*, Techn. Rep., Bell Labs., Murray Hill, N.J., 1968.

[62] MINSKY, M.L., *Computation: Finite and Infinite Machines*, Prentice Hall Inc., Englewood Cliffs, N.J., 1967.

[63] MISRA, J., CHANDI, K.M., *Proofs of Networks of Processes*, IEEE Transactions on Software Engineering, 7,4 (1981) pp. 417-426.

[64] MISUNAS, D.F. (ed.), *Workshop on Data Flow Computer and Program Organization*, Computer Architecture News, SIGARCH, 6,4 (1977), pp. 6-22.

[65] MISUNAS, D.F.(ed.), *Report on the second Workshop on Data Flow Computer and Program Organization*, MIT/LCS/TM-136, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1979.

[66] OLDEHOEFT, A.E., ALLAN, S., THORESON, S., RETNADHAS, C., ZINGG, R.J., *Translation of High level Programs to Data Flow and their Simulated Execution on a Feedback Interpreter*, TR 78-2, Dept. of Comp. Sci., Iowa State University, Ames, 1977.

[67] PARK, D., *The "fairness" problem and nondeterministic computing networks*, in DE BAKKER, J.W., VAN LEEUWEN J.(eds.), *Foundations of Computer science IV. Distributed systems:* Part 2, *Semantics and logic*, Mathematical Centre Tracts 159, Mathematisch Centrum, Amsterdam, 1983, pp. 133-161.

[68] PETERS, F.J., *Tree machines and divide-and-conquer algorithms*, in HÄNDLER, W.(ed.), *Proc. of Compar 81*, LNCS 111, Springer-Verlag, Berlin, 1981, pp. 25-35.

[69] PETERSON, J.L., *Petri Nets*, ACM Computing Surveys, 9,3 (1977), pp. 223-252.

[70] PLAS, A., et.al., *LAU System Architecture: a parallel data-driven processor based on single assignment*, Proc. 1976 International Conf. on Parallel Processing, IEEE, 1976, pp. 293-302.

[71] PLOUFFE, W., ARVIND, GOSTELOW, K.P., *An asynchronous Programming Language and Computing Machine*, TR 114a, Information and Comp. Sci. Dept., University of California, Irvine, 1978.

[72] RITCHIE, D.M., THOMSON, K., *The UNIX Time Sharing System*, CACM, 17,7 (1974), pp. 365-375.

[73] RODRIGUEZ, J.E., *A graph model for Parallel Computation*, MIT/LCS/TR-64, Lab. for Comp. Sci., MIT, Cambridge, Mass, 1969.

[74] ROGERS, Jr. H., *Theory of recursive functions and effective computability*, Mc Graw Hill, New York, N.Y., 1967.

[75] ROZENBERG, G., JANSSENS, D., *On the structure of Node-label Controlled Graph Languages*, Information Sciences, 20 (1980), pp. 191-216.

[76] RUMBAUGH, J., *A data flow multiprocessor*, IEEE Transactions on Computers, C26,2 (1977), pp. 138-146.

[77] SLISENKO, A.O., *Context free grammars as a tool for describing polynomial-time subclasses of hard problems*, Information Processing letters, 14,2 (1982), pp. 52-56.

[78] STONE, H.S., *Parallel Computers*, in STONE, H.S.(ed.), *Introduction to Computer Architecture*, second edition, SRA Computer Science Series, Science Research Associates, Chicago, 1980, pp. 363-425.

[79] THOMPSON, C.D., KUNG, H.T., *Sorting on a Mesh-connected Parallel Computer*, CACM, 20,4 (1977), pp. 263-271.

[80] THORNTON, J.E., *Design of a Computer, The Control Data 6600*, Scott Foresman and Company, Glenview, Illinois, 1970.

[81] THURBER, K.J., *Large Scale Computer Architectures: Parallel and Associative Processors*, Hayden Book Company, Rochelle Park, N.J., 1976.

[82] TIEN CHI CHEN, *Overlap and Pipeline Processing*, in STONE, H.S.(ed.), *Introduction to Computer Architecture*, second edition, SRA Computer Science Series, Science Research Associates, Chicago, 1980, pp. 427-485.

[83] TODD, S., *Algorithm and Hardware for Merge Sort using multiple Processors*, IBM J. Res. Develop., 22,5 (1978), pp. 509-517.

[84] TRELEAVEN, P.C., BROWNBRIDGE, D.R., HOPKINS, R.P., *Data-driven and Demand-driven Computer Architecture*, ACM Computing Surveys, 14,1 (1982), pp. 93-143.

[85] VEEN, A.H., *Reconciling Data Flow Machines and Conventional Languages*, in HÄNDLER, W.(ed.), *Proc. of Conpar 81*, LNCS 111, Springer-Verlag, Berlin, 1981, pp. 127-140.

[86] WADGE, W.W., *An extensional treatment of dataflow deadlock*, in KAHN, G. (ed.), *Semantics of Concurrent Computation*, Proc. Evian, France, LNCS 70, Springer-Verlag, Berlin, 1979, pp. 285-299.

[87] WALLACH, Y., *Alternating sequential/parallel processing*, LNCS 127, Springer_Verlag, Berlin, 1982.

[88] WHITELOCK, P.J., *A conventional Language for Dataflow Computing*, M.Sc. Dissertation, Victoria University of Manchester, 1978.

[89] WIRTH, N., *Modula 2*, Report 36, Institut für Informatik, ETH, Zürich, 1980.

[90] WYLLIE, J.C., *The Complexity of Parallel Computations*, Ph.D. Thesis, TR-79-387, Dept. of Comp. Sci., Cornell Univ., Ithaca, New York, 1979.

INDEX

SAMENVATTING

Dit proefschrift is gewijd aan "dataflow" berekeningen, een bepaald soort parallelle berekeningen. De studie van parallelle berekeningen komt voort uit de behoefte aan snellere computers. Hoofdstuk één bevat een kort overzicht van parallelle computer architecturen en hun onderliggende berekeningsmodellen. Het dataflow berekeningsmodel wordt in wat meer detail behandeld. Een dataflow programma of dataflow net is een gerichte graaf waarin de knopen operaties en de kanten data paden voorstellen. Er wordt niets aangenomen over de tijd die een operatie of een data transport vergt. Omdat het gedrag van knopen en kanten op verschillende manieren gespecificeerd kan worden, zijn er een aantal verschillende dataflow modellen.

Met dataflow netten als onderliggend berekeningsmodel kan men een nieuw soort computers ontwerpen die het intrinsieke parallellisme in dataflow netten uitbuiten. Een bestaande dataflow machine, de "Manchester Dataflow Machine" wordt behandeld. De rest van hoofdstuk één behandelt de ontwikkeling van programmeertalen en algoritmes voor dataflow machines.

Hoofdstuk twee introduceert een elementair dataflow berekeningsmodel. Dit model verschilt van het algemeen geaccepteerde, door Rodriguez en Adams geïntroduceerde model. De operaties zijn meer elementair en het model weerspiegelt het tijdsafhankelijke, niet-functionele gedrag van dataflow machines. Er wordt aangetoond dat voor zogenaamd welgevormde dataflow netten het asynchrone, parallelle gedrag niet leidt tot niet-functionaliteit. Er wordt aangetoond dat het model universele berekeningsmacht heeft, en hiervan worden enkele toepassingen gegeven. Andere berekeningsmodellen zoals "counter" machines, Petrinetten, geheugen cellen en de niet-functionele "matching functions" van de Manchester Dataflow Machine, worden gesimuleerd.

Als mensen programmeren denken ze in berekeningséénheden met de berekeningskracht van procedures en niet in laag niveau operaties zoals de dataflow operaties. Men wil dan ook parallellisme op procedure niveau kunnen uitdrukken. Hoofdstuk drie introduceert een programmeertaal met expliciet parallellisme op procedure niveau, gebaseerd op Kahn's "simple language for parallel programming".

Een programma in executie is een dynamisch veranderend netwerk van processen die met elkaar communiceren via kanalen waarover rijen waarden getransporteerd worden. (Vandaar de naam van de taal: DNP, voor Dynamische Netwerken van Processen). Een belangrijke eigenschap van de taal is dat er geen behoefte is aan globale informatie over de toestand van het net als twee processen met elkaar communiceren of als een deel van het net verandert. Een implementatie van de taal wordt beschreven.

In hoofdstuk vier wordt een aantal algoritmes, geschreven in DNP, gepresenteerd. Deze algoritmes zijn typerend voor dataflow omdat grote datastructuren in stukken worden gebroken en door vele processen tegelijkertijd worden gemanipuleerd. De complexiteit van deze algoritmes wordt geanalyseerd. Tevens worden de beperkingen van DNP behandeld. De belangrijkste stelling is dat niet alle klassen van berekenings grafen gegenereerd kunnen worden. Uitbreidingen van de taal die deze beperking opheffen worden aangegeven. In de rest van hoofdstuk vier wordt een vergelijking gemaakt met standaard complexiteitklassen.

In hoofdstuk vijf worden enkele DNP programma's van hoofdstuk vier correct bewezen. De bewijzen zijn gebaseerd op een semantiek voor DNP die overeenkomt met Kahn's ideeën. De bewijzen zijn lang omdat er met vele details rekening gehouden moet worden. Een tekortkoming van de gebruikte semantiek is dat er alleen uitspraken gedaan kunnen worden over de gehele rij waarden die gedurende een executie van een programma over een kanaal getransporteerd worden. Er kunnen geen uitspraken gedaan worden over de relatieve ordening van bepaalde gebeurtenissen in verschillende processen. Onderzoek aan dit onderwerp wordt elders uitgevoerd.

SUMMARY

This thesis is devoted to dataflow computation, a particular kind of parallel computation. The motivation for parallel computation is the need for faster computing machines. Chapter one gives a short overview of parallel computer architectures and their underlying model of computation. The dataflow model of computation is discussed in some detail. A dataflow program or dataflow net is a directed graph in which the nodes represent processing elements and the edges represent data paths. No assumptions are made about the timing of processing elements or data transports. Various options in specifying the behaviour of the nodes and edges lead to a number of different dataflow models.

Having dataflow nets as the underlying model of computation an unconventional computer architecture can be designed to exploit the intrinsic parallelism of these dataflow nets. An existing and working dataflow machine, the Manchester Dataflow Machine, is discussed. The remainder of chapter one sketches the development of programming languages and algorithms for dataflow machines.

Chapter two introduces an elementary model of dataflow computation. This model differs from the widely accepted dataflow model introduced bij Rodriguez and Adams in that its processing elements are even more primitive and that it mirrors the time-dependent, non-functional behaviour of dataflow machines. It is shown that for so called well-formed dataflow nets the asynchronous, parallel execution mode does not lead to non-functional behaviour. The model is shown to have universal computing power and some applications of this result are given. Other models of computation such as counter machines, Petri-nets, memory cells and the non-functional matching functions of the Manchester Dataflow Machine are simulated.

When programming, people tend to think in terms of units of action with the power of procedures and not of low level operations such as the dataflow primitives. Consequently people want to express parallelism at the procedure level. Chapter three introduces a programming language with explicit parallelism on the procedure level based upon Kahn's simple language for parallel programming. A program in execution is a dynamically changing network of processes communicating with each other via channels, i.e., queues of values

(hence the name of the language: DNP, for Dynamic Networks of Processes). An important aspect of the language is that there is no need for global information about the computation graph while processes are communicating or part of the graph is changing. The implementation of the language is described.

Chapter four presents a number of algorithms written in DNP. These algorithms are believed to be prototypical for dataflow computing in that large datastructures are broken up and manipulated in parallel by many processes. The complexity of the algorithms is analysed. The limitations of DNP are considered. The main theorem is that not all classes of computation graphs can be generated. Ways to overcome this are indicated. In the remainder of chapter four a comparison is made with standard complexity classes.

In chapter five some of the DNP programs of chapter four are proved correct by detailed reasoning. The proofs are based upon a semantics of DNP according to Kahn's ideas. The proofs are long and tedious because many details have to be dealt with. A shortcoming of the semantics used is that only properties of complete sequences of values travelling over channels during a computation can be stated. No statements can be made about the relative ordering of certain events in various processes. Research on these problems is being carried out at a number of institutions.

CURRICULUM VITAE

Naam	:	Böhm, Anton Pedro Willem.
Geboren	:	4 juli 1948, te Rotterdam.
1967	:	Eindexamen HBS-B, Dalton HBS te Rotterdam.
1967-1968	:	Programmeur, Unilever Rotterdam, opleiding bij IBM Amsterdam.
1968-1980	:	Part time systeem analist, Unilever Rotterdam.
1974	:	Ingenieursexamen Wiskunde, afstudeerrichting Informatica, Technische Hogeschool Delft.
1974-1978	:	Wetenschappelijk medewerker, Mathematisch Centrum Amsterdam.
1978-heden:		Wetenschappelijk medewerker, Rijksuniversiteit Utrecht.