High-level synthesis scheduling using fuzzy set techniques

Bierings, J.G.H.E.

Award date:
1993

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

High-Level Synthesis Scheduling Using Fuzzy Set Techniques

J.G.H.E. Bierings

Master Thesis

performed: September 1992 - July 1993
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Ir. M.J.M. Heijligers

Abstract

In High-Level Synthesis some reasonable solutions to the scheduling-problem are known. However, some of these schedulers (e.g. a list-scheduler) do not always offer satisfying results. Therefore it was desirable to design an algorithm to overcome these problems.

In this report a *feasibility-(pre-)scheduler*, based on *fuzzy set techniques*, is presented. In this approach, all possible start-positions of an operation are regarded as members of a fuzzy set. To each member of these fuzzy sets an initial membership-grade is assigned. To be able to translate properties of the data-flow graph into membership-grades for the elements in the fuzzy set, fuzzy membership-functions are introduced. Then, starting with the initial membership-grades, in an iterative process old membership-grades are translated into new membership-grades.
Given a data-flow graph with a time- and a resource-constraint, the fuzzy scheduler assigns to each possible start-position of an operation a membership-grade. However, the operations are not fixed yet !

To perform the actual scheduling (i.e. fixing the operations), the results of the algorithm can be used in two ways :

- By a *greedy-feasibility scheduler*, which is a scheduler, specially tuned on using membership-grade to make a feasible schedule
- By various other schedulers (e.g. list-schedulers, force-directed schedulers), as a kind of initialization

The fuzzy (pre-)scheduler is fast and produces good results. Furthermore, the algorithm has shown to converge to a solution, even if the membership-grades are initialized at random. However some work has to be done, to make the fuzzy scheduler applicable in a chip-design environment.

The chapters 1 and 2 give a short introduction to High-Level Synthesis and the scheduling-problem. In chapter 3 some relevant topics of fuzzy sets are discussed. Chapter 4 discusses how the fuzzy set techniques are applied to the scheduling-problem. After these definitions, the actual fuzzy algorithm is presented, followed by an example. In chaper 5 the results of the fuzzy algorithm, applied to some examples, are presented. Furthermore, the *greedy-feasiblity scheduler* is outlined and some recommandations for future work are done. Finally, chapters 6 presents the conclusions.

Contents

Chapter 1

Introduction

At the Design Automation Section of the Eindhoven University of Technology, software tools are being developed for the design of integrated circuits. One of these tools is a *Silicon Compiler*, which translates a functional description for some digital system into a chip layout. The silicon compiler can be splitted in three parts (see figure 1.1) : *High-*

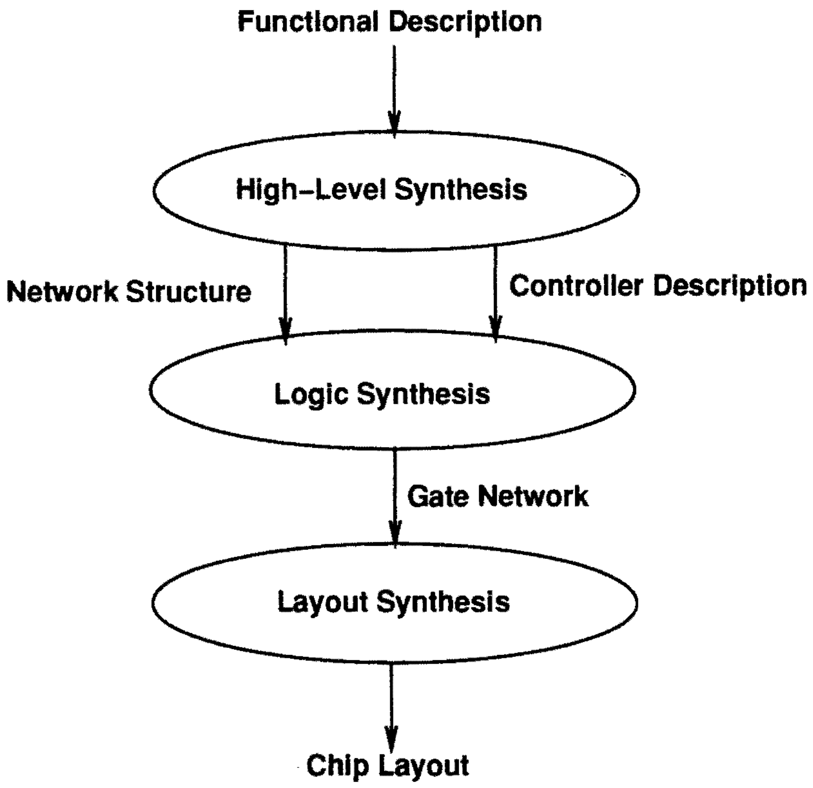

Figure 1.1: Silicon Compiler Overview.

Level Synthesis, *Logic Synthesis* and *Layout Synthesis*. The High-Level Synthesis part transforms the functional description into a description of a network of modules (e.g. multipliers, adders, etc.) and a corresponding (network-)controller. The Logic Synthesis transforms these results into a network of gates. Finally, the Layout Synthesis creates a chip layout from the gate network.

This report is concerned with a problem in High-Level Synthesis, called *scheduling*. In chapter 2, this problem is defined more formally. Although previous work in the design automation section of the Eindhoven University of Technology already offered good solutions for this problem ([Heijligers91]), they also showed to have a few short-comings. Therefore, it was desirable to develop an algorithm, based on fuzzy set techniques, that could provide some kind of initial schedule. Eventually, by using these results, the above mentioned (schedule) algorithms, should overcome their short-comings and offer better results.

Chapter 2

High-Level Synthesis

2.1 System Overview

Within the high-level synthesis process, the following steps can be distincted (see figure 2.1). An object used to interchange information between the various steps in

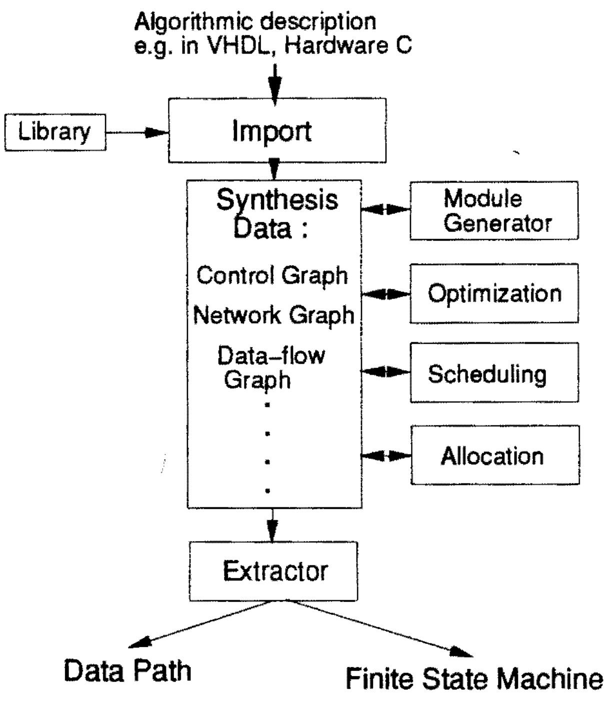

Figure 2.1: High-Level Synthesis system overview.

the high-level synthesis process, is the data-flow graph. The main advantages to use a data-flow graph for synthesis are :

- independency of input description languages used
- detection of concurrency in the description
- no use of variables, but only use of values

Take for example, the following description :

```
process exgraph(a0,a1,a2,a3,f0,f1)
   in port a0, a1, a2, a3;
   out port f0, f1;
{
   integer x0, x1;
   integer y1;
   x0 = a0 + a1;
   x1 = a2 - a3;
   y1 = x0 + x1;
   f0 = x0 * y1;
   f1 = x1 + y1;
}
```

This functional description can be mapped onto the data-flow graph as shown in figure 2.2. With respect to figure 2.2, there are a number of problems to be solved :

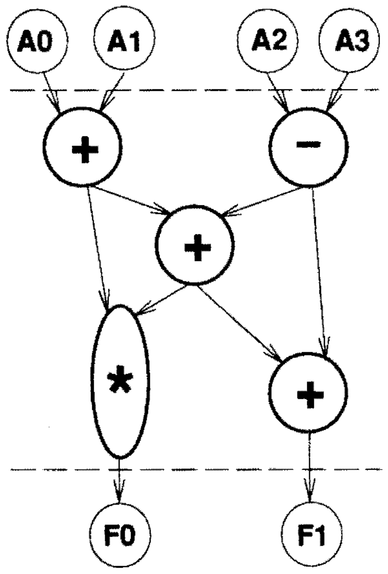

Figure 2.2: Data-flow graph, representing functional description.

- Which and how many modules can be used ?
- At what time must a node be executed ?
- On what module in the data-path will a node be executed ?

At the Eindhoven University of Technology, in the NEAT-system the design-flow in High-Level Synthesis is as follows :
A module-generator takes care of the first point. By investigating the data-flow graph and an associated time-constraint T_{max}, the generator checks which operation-types are used. It then creates from a library a set of modules to be used in the synthesis-process (A library is a database containing a set of modules which can execute one or more operation types. E.g. : <ALU: -,+,*,/,or,and>,<adder:+>).

The scheduler takes care of the second point. It assigns to each node in the graph a cycle-step, in which an operation should start. Depending on the type of scheduler (see section 2.2.4), several constraints occur. For instance, the scheduler might have to know the number of modules (of a certain type) and/or the number of cycle-steps it is allowed to use. Note, the results of the scheduler and the module-selector depend heavily on each other. An illustration of this dependency is given in figure 2.3.

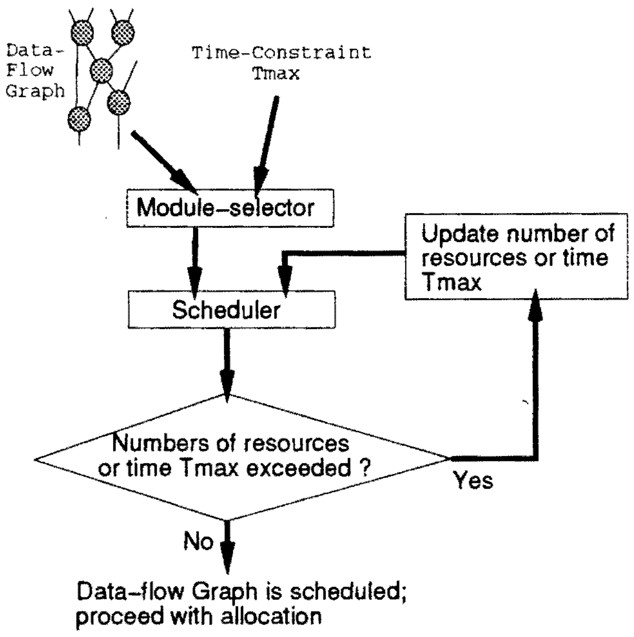
Data-Flow Graph
Time-Constraint Tmax
Module-selector
Update number of resources or time Tmax
Scheduler
Numbers of resources or time Tmax exceeded ?
Yes
No
Data-flow Graph is scheduled; proceed with allocation

Figure 2.3: Interaction between module-selector and scheduler.

The last point is solved by a binder. The binder assigns to each node in the graph a hardware module. Hence, the binder needs to know in which cycle-step nodes are executed, in order to be able to prevent conflicts in the module-usage. Furthermore, register-allocation and interconnect-generation are also performed in this phase. In register-allocation registers are assigned to values which have to be stored during one or more cycles. Interconnect-generation is performed after binding and register-allocation, and determines the interconnection units (i.e. wires and interconnection units like multiplexers and demultiplexers).

In the next section, an introduction and some basic definitions regarding the *scheduling-problem* will be given.

2.2 Scheduling

In short, a scheduler assigns to each node in the data-flow graph one or more cycle steps, that is, during these cycles the node is executed. A scheduler is subjected to constraints : a *precedence-constraint* and depending on the type of scheduler (see section 2.2.4), a *resource-* and/or a *time-constraint*. The precedence-constraint means that a node cannot be executed earlier than its predecessors and cannot be executed later than its successors; this constraint is derived from the data-flow graph.

A data-flow graph is a directed graph. The nodes of the data-flow graph represent operations, the edges model the transfer of values (i.e. tokens) between these nodes. A node can be executed only, if there are values (tokens), on all its incoming edges. In the high-level synthesis process two other kinds of graphs are associated with the data-flow graph : the control-graph and the network-graph. In this report attention will be paid only to data-flow graphs.

2.2.1 The Data-Flow Graph

A data-flow graph is a directed graph, consisting of nodes and edges. This can be stated formally as :
A data-flow graph is a graph $G = (V, E)$, in which :

- *V is a finite set of nodes*
- *$E \subseteq V \times V$ is a set of directed edges*

In the graph, there exist various kinds of node types, each having its own semantics. There are *operation nodes, input- and output nodes, constant nodes, branch- and merge nodes*, etc. In this report only operation- and input/output-nodes will be used. An *operation node* represents operations like *,-,+, /, and boolean operators like *and,or,<,>*. The input/output-nodes are used to interface with the outside world, i.e. input-values are assigned to the input-nodes, output-nodes return the computed values. In the data-flow graph $G = (V, E)$, the function τ performs the mapping of nodes onto types :

$$\tau : V \rightarrow Type \tag{2.1}$$

with *Type* the set of *(node-)types*.

For each node in the graph two special sets of nodes associated to that node, can be defined :

$$succ(v_i) = \{v_j \in V \mid (v_i, v_j) \in E\} \tag{2.2}$$

representing the set of all direct successors of a node $v_i \in V$, and

$$pred(v_j) = \{v_i \in V \mid (v_i, v_j) \in E\} \tag{2.3}$$

representing the set of all direct predecessors of a node $v_j \in V$. Note, there also exist several kinds of edges (see [Eijndhoven91]), however we will only be concerned with *data edges.*

2.2.2 The library

As outlined in section 2.1, the scheduler uses a library to obtain information about operation-types in the data-flow graph. To be able to schedule, the scheduler needs to know the execution time of all the operations in the graph. For convenience it is assumed that every operation has one unique module upon which it will be executed. Hence, every operation in the data-flow graph has a unique execution-time. This can be stated formally as :

Operation mapping
Let $G = (V, E)$ be a data-flow graph, L be a library.
$\xi : V \rightarrow L$ is a mapping from operations to modules.

Delay
Let $G = (V, E)$ be a data-flow graph, L be a library.
$\delta : V \times L \rightarrow \mathbb{R}$ is a function decribing the time a module needs to execute a node.

Example of a library
Let L be a library : L={<ALU:-,+,*,/,*or*, *and*>,<adder:+,->,<I/O:get,put>}

2.2.3 Some more definitions

Given a data-flow graph, the scheduler has to decide which node is executed at what time. The time is given by the interval $[0..T_{max}]$, where T_{max} is the moment at which all nodes in the graph must have been executed. The integer interval $[0..T_{max} - 1]$ represents the set of cycle-steps, involved in scheduling the graph. An example of time and cycle-steps is given in figure 2.4, where [0..4] denotes the time interval and [0..3]

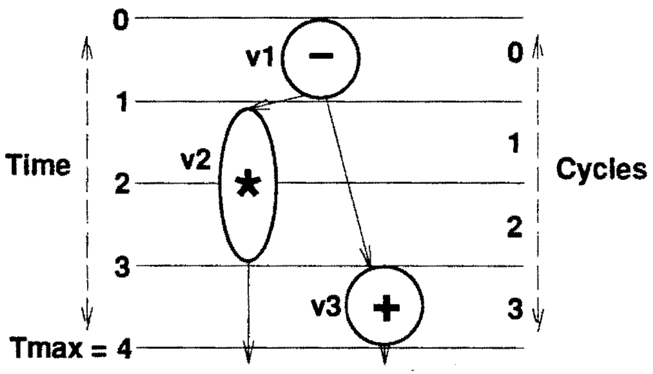

Figure 2.4: Illustration of time and cycle-steps.

denotes the integer interval of cycle-steps.

The use of cycles implies the use of some (central) clock, operating at a certain frequency, and hence the regarded system is one of a synchronous kind. Furthermore, it is assumed that all modules are related to this specified clock-frequency, i.e. the delay of the modules is specified in the number of cycle-steps.
In figure 2.5, two special situations are drawn : *multi-cycling* and *chaining*. In words :
Multi-cycling : the delay of a module is bigger than one cycle-step; the corresponding operations will occupy more than one cycle.
Chaining : the delay of a module is smaller than one cycle-step; sometimes more than one related operations can be executed in one cycle-step.

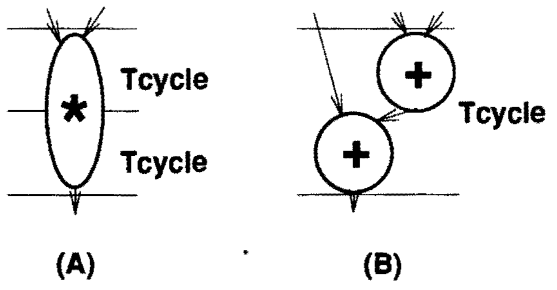

Figure 2.5: (a) Multi-cycling (b) Chaining.

Finally, a module used can be a *pipelined* module. Pipelined modules can start a new execution before the previous execution has finished. This means that the module can contain more than one data-value at the same time. However, in this report, pipelining is not considered.

Thanks to the precedence-constraint, to each node two values can be assigned :

- An *ASAP* (As Soon As Possible) value, representing the earliest time at which execution of a node can be started.
- An *ALAP* (As Late As Possible) value, representing the latest time at which execution of a node must be finished.

Stated formally :
Given the data-flow graph $G = (V, E)$, node $v_i \in V$ and time-constraint T_{max} :

$$asap(v_i) = \begin{cases} 0 & \text{if } pred(v_i) = \emptyset \\ max_{v_j \in pred(v_i)}[asap(v_j) + \delta(v_j)] & \text{if } pred(v_i) \neq \emptyset \end{cases} \tag{2.4}$$

$$alap(v_i) = \begin{cases} T_{max} & \text{if } succ(v_i) = \emptyset \\ min_{v_j \in succ(v_i)}[alap(v_j) - \delta(v_j)] & \text{if } succ(v_i) \neq \emptyset \end{cases} \tag{2.5}$$

Now, the pair $[asap(v), alap(v)]$ denotes the interval in which a node v can be scheduled, and is called the *execution-interval* of node v ($EI(v)$). Likewise, the pair $[asap(v), alap(v) - \delta(v) + 1]$ denotes the interval, in which the execution of a node v can be started and is called the *start-interval* of node v ($SI(v)$). Corresponding to $SI(v)$, $C_v = [asap(v), alap(v) - \delta(v)]$ denotes the integer interval of cycle-steps of the start-interval of v.

With respect to figure 2.4, the following intervals can be distincted :
$EI(v_1) = [0,2]$, $SI(v_1) = [0,2]$, $C_{v_1} = \{0,1\}$
$EI(v_2) = [1,4]$, $SI(v_2) = [1,3]$, $C_{v_2} = \{1,2\}$
$EI(v_3) = [1,4]$, $SI(v_3) = [1,4]$, $C_{v_3} = \{1,2,3\}$

with : $\delta(v_1) = \delta(v_3) = 1$ cycle, and $\delta(v_2) = 2$ cycles.

2.2.4 Different kinds of scheduling

Rougly spoken, three kinds of scheduling can be distincted :

- Time-constrainted scheduling
- Resource-constrainted scheduling
- Feasible scheduling

With the first one, time-constrainted scheduling, a time-constraint T_{max} is given. This means, the data-flow graph must be fully executed before this time. It is assumed that the scheduler has the disposal of an infinite amount of modules (of any kind).

With the second one, resource-constrainted scheduling, a resourc-constraint as a limited amount of modules (of all types) is given, however there is no time-constraint. This means, that the scheduler can use as much time as it needs to make a schedule.

The third one, feasible scheduling, in fact is a combination of time-constrainted- and resource-constrainted-scheduling. In this case, a time-constraint, as well as a resource-constraint are given. The scheduler is then asked to create a feasible schedule.

In figure 2.6, an example of a graph is given, which is scheduled in the three ways as discussed above.

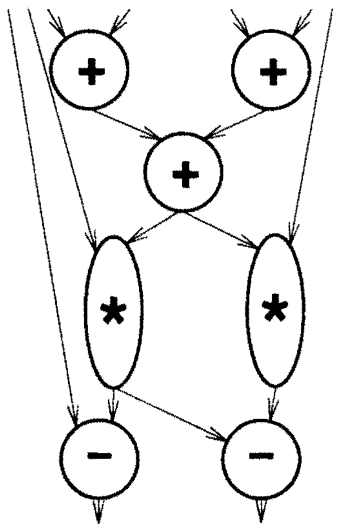

Figure 2.6: Example of a data-flow graph to be scheduled.

For the examples shown in the figures 2.7 to 2.9, the assumption is made that the modules used are chosen from the library :
L={<multiplier:*>,<adder:+>,<subtractor:->}.
For figure 2.7 this implies that given the time-constraint T_{max}, the graph can be scheduled using 2 adders, 2 multipliers and 2 subtractors.
Likewise, the resource-constraint chosen in figure 2.8, implies the graph can be scheduled in 8 cycles.

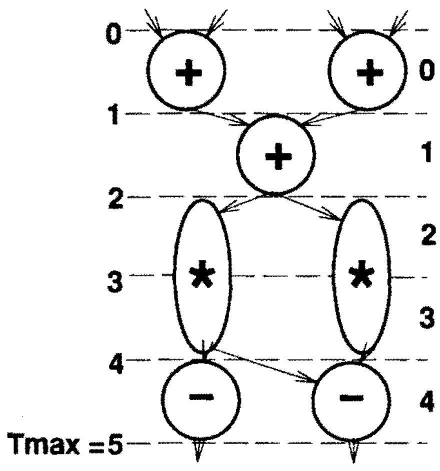

Figure 2.7: Time-constrainted scheduling, with $T_{max} = 5$.

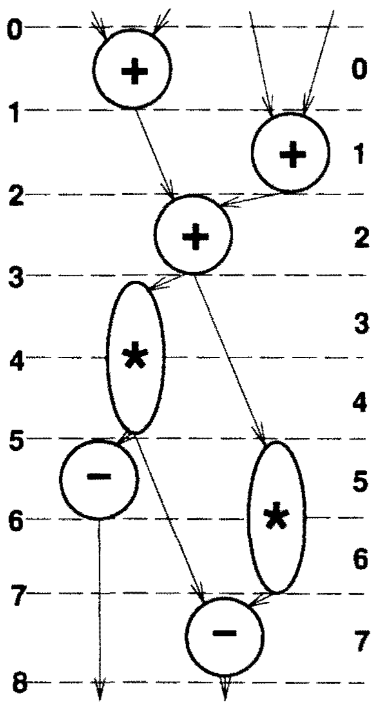

Figure 2.8: Resource-constrainted scheduling, with 1 adder, 1 multiplier and 1 subtractor.

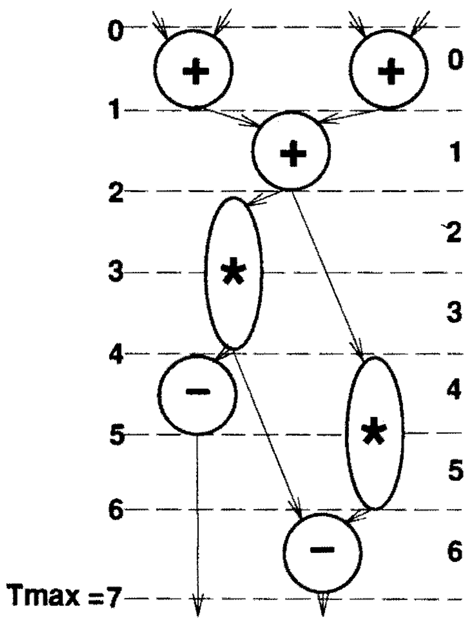

Figure 2.9: Feasible scheduling, with $T_{max} = 7$ and 2 adders, 1 multiplier and 1 subtractor.

Chapter 3

Fuzzy Sets

3.1 History

In 1965, Zadeh introduced the idea of *fuzzy sets* [Zad65]. It was an attempt to adapt the concepts of fuzzy boundaries to science. Since then, scientists have had the disposal of a tool to cope with formulations like *a tall guy*, *a red apple* or *middle-aged people*.
Although fuzzy sets may sound a little bit "fuzzy", the opposite is true : The fuzzy-set-theory has got a well-defined mathematical foundation with fine properties. On top of that, the basic concept of the fuzzy-set-theory can be understood quite easily.
Since its introduction, the fuzzy-set-theory has assumed enormous proportions over a wide range of applications. Nowadays, fuzzy sets and fuzzy logic (a logic based on fuzzy sets) are succesfully applied in control engineering, speech-recognition and pattern-recognition, to mention a few.
Furthermore it is remarkable that *fuzzy*-systems succeed in situations, where systems based on the *non-fuzzy*-theories, like classical logic and classical-control-theory fail. The other way round, *non-fuzzy*-systems can handle a lot of situations wherein *fuzzy*-systems would act as a car with square wheels.

In the next section, a brief introduction to fuzzy sets will be given, so that the reader gets an impression of the properties and possibilities of fuzzy sets.

3.2 A brief introduction to Fuzzy Sets

3.2.1 Introduction

In classical logic, all reasoning is based on two values : *true* or *false*. Problems, like the one below, can easily be described :

Given the set $\mathcal{S} = \{x \in \mathbb{N} \mid x \leq 25\}$, and the function $\mathcal{F}(x)$ with :

$$\mathcal{F}(x) = \begin{cases} 1 & \text{if } x \leq 25 \\ 0 & \text{if } x > 25 \end{cases}$$

Then : $\mathcal{F}(24) = 1$, $\mathcal{F}(25) = 1$ and $\mathcal{F}(26) = 0$.

For a lot of situations this seems to be adequate. However in trying to describe a set like *young people*, classical logic cannot provide a simple method (Or not even a method at all !!), to decide whether an age of 25 is *young* or *old*. And what to say about someone with an age of 26, when 25 years is assumed to be the border of *young* and *old* ? As a consequence of this assumption, someone who's having his/her 25-th birthday, will get *old* in less then a second !!! It may be obvious that this is not realistic.

Fortunately, fuzzy sets provide a simple way to cope with this problem. As a matter of fact, getting older, in most cases, is a smooth process : a man gets a little bit older (or a little bit less young) every day.
Therefore, it's reasonable to use a function $\mathcal{M}_{young}(x)$, as defined below ($\mathcal{M}_{young}$ and $\mathcal{F}$ are plotted in figure 3.1) :

$$\mathcal{M}_{young}(x) = \begin{cases} 1 & \text{if } 0 \leq x < 20 \\ \frac{1}{1+\frac{1}{10}(x-20)} & \text{if } x \geq 20 \end{cases} \tag{3.1}$$

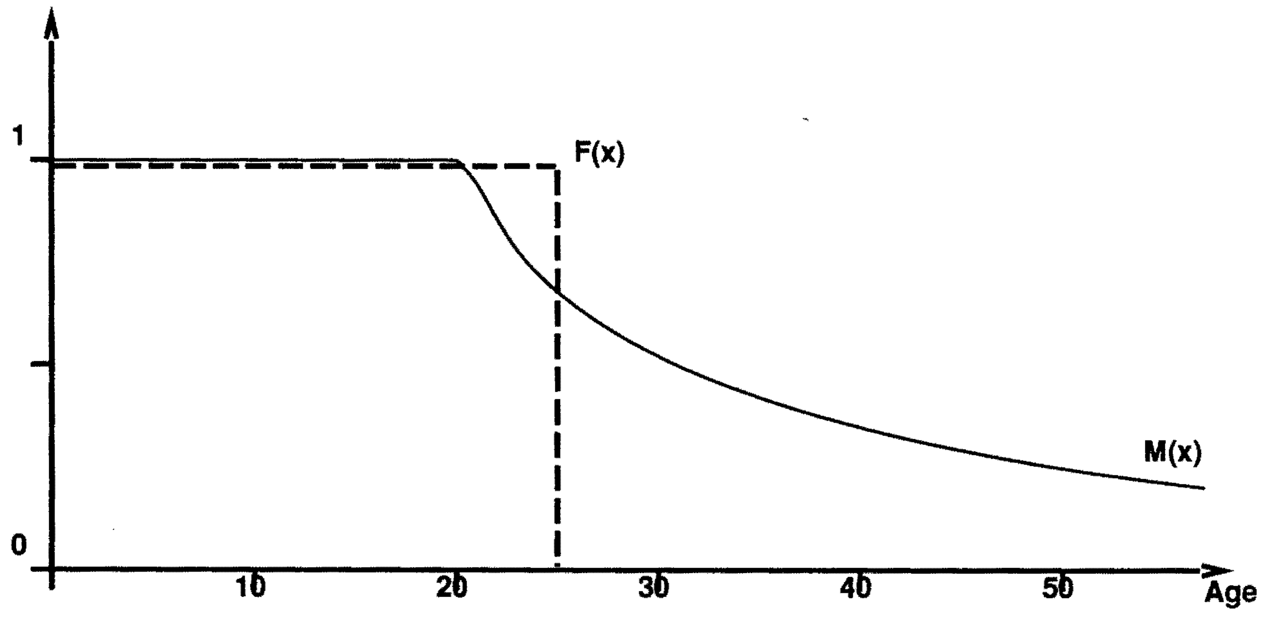

Figure 3.1: Plots of $\mathcal{F}$ and $\mathcal{M}_{young}$.

As can can be seen from the definition of $\mathcal{M}_{young}$, it's a function mapping $\mathbb{N} \mapsto [0,1]$. Hence $\mathcal{M}_{young}$ produces :
$\mathcal{M}_{young}(15) = 1$, $\mathcal{M}_{young}(24) \approx 0.71$, $\mathcal{M}_{young}(25) \approx 0.67$, $\mathcal{M}_{young}(26) \approx 0.63$ and $\mathcal{M}_{young}(80) \approx 0.14$.
The values returned by $\mathcal{M}_{young}$ correspond to our (intuitive) idea of getting older :
An age of 15 certainly is *young*, corresponding to the value 1; on the contrary, an age of 80 can hardly said to be *young*, corresponding to the value 0.14.

The function $\mathcal{M}_{young}$ is called a *membership-function* of the fuzzy set *young*. In the next paragraph some basic properties of fuzzy sets will be defined.

3.2.2 Basic Properties

Zadeh [Zad65] defines a fuzzy set as a mapping of the set $\mathcal{X}$ onto the unit interval [0,1]. The following is involved.

Let $\mathcal{X}$ be a set. The fuzzy set $\mathcal{S}$ in $\mathcal{X}$ is defined as :

$$\mathcal{S} = \mathcal{X} \times [0,1] \tag{3.2}$$

with the membership-function $\mu_S : \mathcal{X} \to [0,1]$, which associates to each $x \in \mathcal{X}$ a real number in the interval $[0,1]$.

The number $\mu_S(x)$ is called the ***grade of membership*** of x in the fuzzy set $\mathcal{S}$. The closer the value of $\mu_S(x)$ is to 1, the greater is the membership grade of x in $\mathcal{S}$. Obviously, the membership-function is a generalization of the characteristic function of ordinary set theory, which takes on the two values : 1 for elements belonging to the set, and 0 for elements not belonging to the set $\mathcal{S}$.

For a set $\mathcal{X}$ the fuzzy set $\mathcal{S}$ is written as the set of tuples :

$$\mathcal{S} = \{(x, \mu_S(x))\}.$$

The results of equation 3.1 can now be expressed as :
$\mathcal{S} = \{\ldots,(15,1),\ldots,(24,0.71),\ldots,(25,0.67),\ldots,(26,0.63),\ldots,(80,0.14),\ldots\}$

In the general case the choice of a membership-function $\mu_S(x)$ is subjective, and based on information available in each individual situation.

In the construction of the theory of fuzzy sets, the foregoing is a step to the definition of operations on fuzzy sets. Note that only the most important operations on fuzzy sets are defined here, in particular those operations that are significant with respect to the next chapter. Below the operations are presented :

Empty Set
A fuzzy set $\mathcal{S}$ is said to be empty ($\mathcal{S} = \emptyset$),
i.f.f. for all $x \in \mathcal{X}$ holds $\mu_S(x) = 0$.

Equivalence
Two fuzzy sets $\mathcal{A}$ and $\mathcal{B}$ are said to be equivalent ($\mathcal{A} \equiv \mathcal{B}$),
i.f.f. for all $x \in \mathcal{X}$ holds $\mu_A(x) = \mu_B(x)$.

Complement
The fuzzy set $\overline{S}$ is the complement of the fuzzy set $\mathcal{S}$
i.f.f. $\mu_{\overline{S}}(x) = 1 - \mu_S(x)$.

Union
The union $\mathcal{A} \cup \mathcal{B}$ of the two fuzzy sets $\mathcal{A}$ and $\mathcal{B}$, is defined as the least fuzzy set containing both sets $\mathcal{A}$ and $\mathcal{B}$. The membership-function of the set $\mathcal{A} \cup \mathcal{B}$ is defined by the expression :

$$\mu_{\mathcal{A}\cup\mathcal{B}} = \max(\mu_A(x), \mu_B(x)) \tag{3.3}$$

Intersection
The intersection $\mathcal{A} \cap \mathcal{B}$ of the two fuzzy sets $\mathcal{A}$ and $\mathcal{B}$, is defined as the greatest fuzzy set that is simultaneously a subset of both of these sets. The membership-function of the set $\mathcal{A} \cap \mathcal{B}$ is defined by the expression :

$$\mu_{\mathcal{A}\cap\mathcal{B}} = \min(\mu_A(x), \mu_B(x)) \tag{3.4}$$

Normalization
Two forms of normalization can be distincted :

- *Normalization on greatest membership-grade of all elements in the set.*
- *Normalization on cardinality of the fuzzy set.*

The fuzzy set $\mathcal{Y}'$, which is normalized on the greatest membership-grade of all elements of the fuzzy set $\mathcal{Y}$ (in $\mathcal{X}$), can be expressed as

$$\mathcal{Y}' = \{(a,b) \mid a \in \mathcal{X} \wedge b = \frac{\mu_Y(a)}{K}\} \tag{3.5}$$

With :

$$\mathcal{X} \text{ is a set, and } K = \mu_Y(h), \text{ with } \forall_{y\in X}[(\mu_Y(y) \leq \mu_Y(h)) \wedge h \in \mathcal{X}]$$

The fuzzy set $\mathcal{Y}'$, which is normalized on the cardinality of the fuzzy set $\mathcal{Y}$ (in $\mathcal{X}$), can be expressed as :

$$\mathcal{Y}' = \{(a,b) \mid a \in \mathcal{X} \wedge b = \frac{\mu_Y(a)}{|\mathcal{Y}|}\} \tag{3.6}$$

$$\text{with : } |\mathcal{Y}| = \sum_{y\in X} \mu_Y(y) \tag{3.7}$$

Of course far more operations can be defined, however the operations discussed above are sufficient for this report. More operations on fuzzy sets and formal proofs of the operations, can be found in [Klir88] and [Zad65].

Chapter 4

Fuzzy Scheduling

4.1 Introduction

In this chapter a schedule algorithm based on fuzzy set techniques will be presented. As discussed in chapter 2, scheduling a data-flow graph $G = (V, E)$ is the process of assigning a cycle to each node, in which it should start.
In short, the problem to be solved is :
Given a data-flow graph $G = (V, E)$, a time-constraint T_{max} and a resource-constraint for the number of resources.
Question :
Determine for each cycle in the start-interval of a node, a membership-grade, such, that eventually, starting a node in the cycle with the highest membership-grade results in a feasible schedule.

4.2 Fuzzy Sets and Feasible Scheduling

In this section the fuzzy-set theory of the previous chapter will be matched with the scheduling-problem as discussed in chapter 2.

4.2.1 Set-Definitions

With respect to the definitions of the *start-interval* SI and the *execution-interval* EI in section 2.2.3, two kinds of fuzzy sets can be distinguished :

(A) Node oriented
This kind of fuzzy set associates a membership-grade to each cycle in the start-interval of a node. The node-oriented fuzzy set S_v can be defined as :

$$S_v = C_v \times [0, 1] \qquad (4.1)$$

With : v a node of the data-flow-graph.
C_v the set of cycles of the start-interval of v.

(B) Cycle oriented

This kind of fuzzy set associates a membership-grade to each node from the set of nodes, that can be mapped onto the same module-type and which can start in the considered cycle. The cycle-oriented fuzzy set $M_c(modtype)$ can therefore be defined as :

$$M_c(modtype) = T_c(modtype) \times [0,1] \tag{4.2}$$

and

$$T_c(modtype) = \{v \in V \mid c \in C_v \ \wedge \ modtype \in R_v\} \tag{4.3}$$

with : V the set of all nodes in the data-flow-graph.
C_v the set of cycles of the start-interval of v.
R_v the set of module-types where v can be mapped onto.

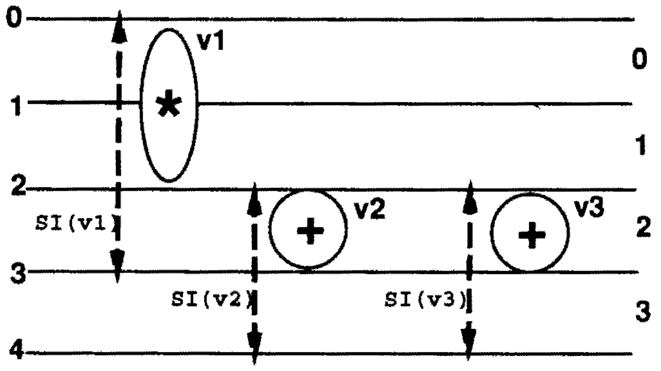

Figure 4.1: Node- and Cycle-oriented fuzzy sets.

Hence, in figure 4.1, the following node- and cycle-oriented fuzzy sets can be distincted (note that the membership-grades of the nodes in these sets are chosen arbitrarily) :

Node-oriented fuzzy sets :

$S_{v_1} = \{(0,0.5),(1,1),(2,0.8)\}$
$S_{v_2} = \{(2,0.2),(3,1)\}$
$S_{v_3} = \{(2,1),(3,0.7)\}$

Cycle-oriented fuzzy sets :

$M_0(adder) = M_1(adder) = \emptyset$
$M_2(adder) = \{(v_2,0.2),(v_3,1)\}$
$M_3(adder) = \{(v_2,1),(v_3,0.7)\}$
$M_0(ALU) = \{(v_1,0.5)\}$
$M_1(ALU) = \{(v_1,1)\}$
$M_2(ALU) = \{(v_1,0.8),(v_2,0.2),(v_3,1)\}$
$M_3(ALU) = \{(v_2,1),(v_3,0.7)\}$

Given :
(i) $\delta(v_1) = 2,\ \delta(v_2) = \delta(v_3) = 1.$
(ii) $R_{v_1} = \{ALU\},\ R_{v_2} = R_{v_3} = \{ALU, adder\}.$

4.2.2 Distorsion

Now the fuzzy sets with respect to the scheduling-problem are defined, the term *distorsion* can be introduced. Roughly spoken, *distorsion* is a measure for the impact a certain node has on another node, when the first one would be started in one of the cycles of its start-interval. Obviously, if a node would be started later, it causes distorsion ($DIST_{down}$) with respect to all its successors; in the same way, a node causes distorsion ($DIST_{up}$) with respect to all its predecessors, if it would be started earlier. Remind, the aim of scheduling is to assign to each node a cycle, in which it is allowed to start. So, it is obvious that if a node causes a lot of distorsion with respect to its predecessors or successors, letting the node start in that cycle is not a fortunate decision. A node could be started best in a cycle, in which it causes the smallest distorsion to other nodes as possible. In figure 4.2 can be seen, that if v_4 is started in cycle 1, v_7 still can start in

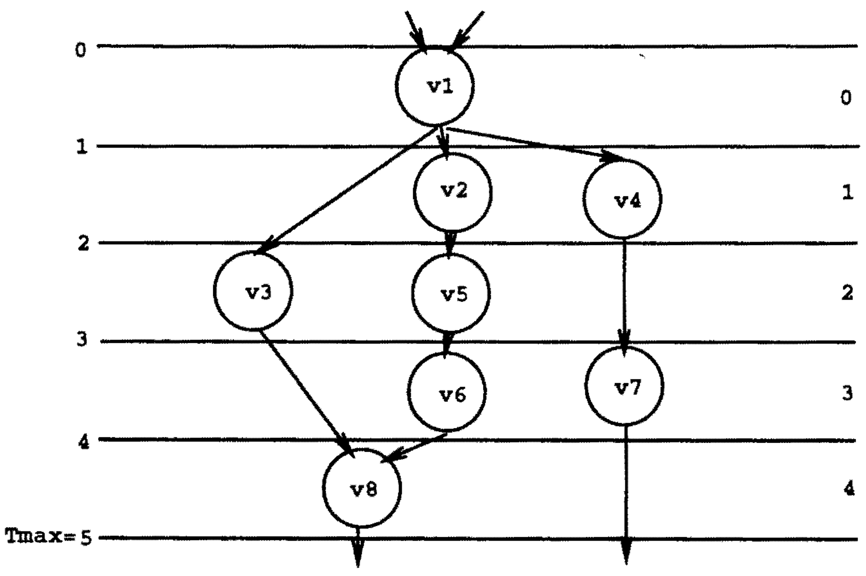

Figure 4.2: Example of distorsion.

all $c \in C_{v_7}$. However if v_4 would be started in cycle 3, v_7 must be started in cycle 4, because otherwise it would violate the time-constraint T_{max}. The other way around, if v_7 is started in cycle 4, this leaves to v_4 the choice to start in either cycle of C_{v_4}. Starting v_7 in cycle 1, forces v_4 to start in cycle 1. In general, starting a node earlier or later, will affect the start of predecessors and successors respectively. It can be stated, that, the greater the distorsion a node causes in either direction, the worse the decision becomes to start the node in the regarded cycle. In fact, a situation arises as plotted in figure 4.3. In this figure, *quality* denotes the quality of the decision to start a node in an arbitrary cycle, as a function of the distorsion caused by this node, with respect to other nodes : the closer the *quality* is to 1, the better the decision; the closer the *quality* is to 0, the worse the decision.

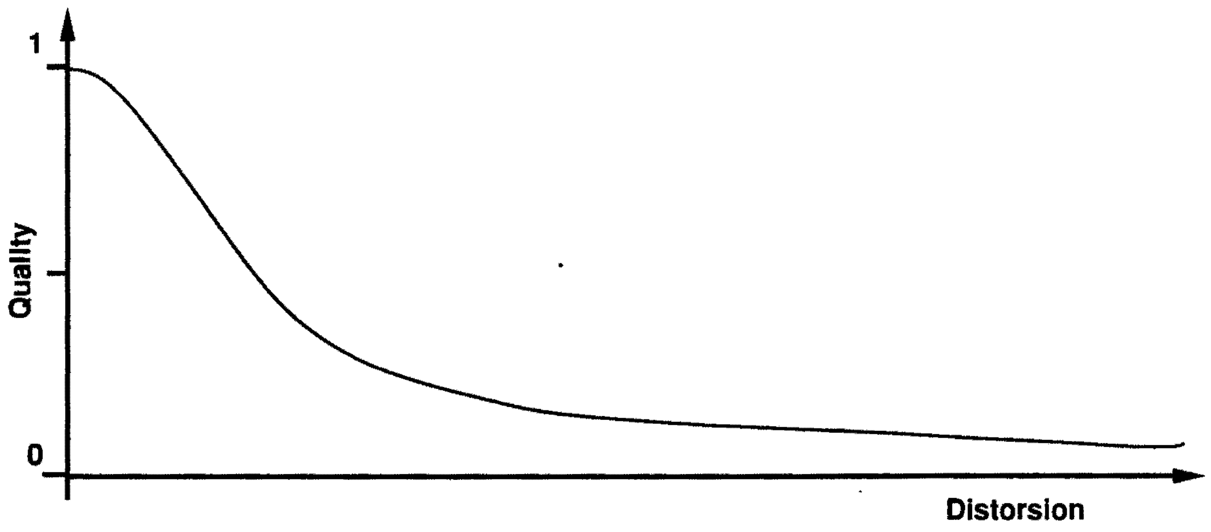

Figure 4.3: Distorsion-Quality plot.

Note that the situation in figure 4.2 can be stated formally as :

$$DIST_{down}(v_i, c) = \sum_{v_j \in succ(v_i)} \sum_{c_j=0}^{c+\delta(v_i)-1} \mu_{v_j}(c_j) + \sum_{v_j \in succ(v_i)} DIST_{down}(v_j, c + \delta(v_i)) \qquad (4.4)$$

with : $c_j \in C_{v_j}$, $c \in C_{v_i}$ and $(v_i, v_j) \in E$.
If $succ(v) = \emptyset$ then $DIST_{down}(v,c) = 0$, for all $c \in C_v$.
$\mu_{v_j}(c_j)$ representing the membership-grade of cycle c_j in S_{v_j}.

$$DIST_{up}(v_j, c) = \sum_{v_i \in pred(v_j)} \sum_{c_i=c-\delta(v_i)+1}^{T_{max}-1} \mu_{v_i}(c_i) + \sum_{v_i \in pred(v_j)} DIST_{up}(v_i, c - \delta(v_i)) \qquad (4.5)$$

with : $c_i \in C_{v_i}$, $c \in C_{v_j}$ and $(v_i, v_j) \in E$.
If $pred(v) = \emptyset$ then $DIST_{up}(v,c) = 0$, for all $c \in C_v$.
$\mu_{v_i}(c_i)$ representing the membership-grade of cycle c_i in S_{v_i}.

Assuming, the membership-grades of all $c_i \in C_{v_i}$, for all v in the data-flow graph of figure 4.2 are set to 1 initially, the distorsions $DIST_{up}(v,c)$ and $DIST_{down}(v,c)$ then become :

$DIST_{down}(v_1,0) = DIST_{up}(v_1,0) = 0,$ $DIST_{down}(v_8,4) = DIST_{up}(v_8,4) = 0,$
$DIST_{down}(v_2,1) = DIST_{up}(v_2,1) = 0,$ $DIST_{down}(v_3,1) = DIST_{up}(v_3,1) = 0,$
$DIST_{down}(v_5,2) = DIST_{up}(v_5,2) = 0,$ $DIST_{down}(v_3,2) = DIST_{up}(v_3,2) = 0,$
$DIST_{down}(v_6,3) = DIST_{up}(v_6,3) = 0,$ $DIST_{down}(v_3,3) = DIST_{up}(v_3,3) = 0,$

$DIST_{down}(v_7,2) = 0,$ $DIST_{up}(v_4,1) = 0,$
$DIST_{down}(v_7,3) = 0,$ $DIST_{up}(v_4,2) = 0,$
$DIST_{down}(v_7,4) = 0,$ $DIST_{up}(v_4,3) = 0,$

$$\begin{aligned}
DIST_{down}(v_4,1) &= 0 + DIST_{down}(v_7,2) = 0+0 = 0,\\
DIST_{down}(v_4,2) &= (\mu_{v_7}(0)+\mu_{v_7}(1)+\mu_{v_7}(2)) + DIST_{down}(v_7,3) = (0+0+1)+0 = 1,\\
DIST_{down}(v_4,3) &= (\mu_{v_7}(0)+\mu_{v_7}(1)+\mu_{v_7}(2)+\mu_{v_7}(3)) + DIST_{down}(v_7,4) =\\
&\quad (0+0+1+1)+0 = 2,
\end{aligned}$$

$$\begin{aligned}
DIST_{up}(v_7,4) &= 0 + DIST_{up}(v_4,3) = 0+0 = 0,\\
DIST_{up}(v_7,3) &= (\mu_{v_4}(4)+\mu_{v_4}(3)) + DIST_{up}(v_4,2) = (0+1)+0 = 1,\\
DIST_{up}(v_7,2) &= (\mu_{v_4}(4)+\mu_{v_4}(3)+\mu_{v_4}(2)) + DIST_{up}(v_4,1) = (0+1+1)+0 = 2.
\end{aligned}$$

The distorsion a node causes, with respect to its successors (predecessors), is equal to the sum of the membership-grades of the cycles, in which the successors (predecessors) (and the successors(predecessors) of the successors (predecessors)) are not allowed to start any more. Note that, although a node has a successor and a predecessor, it does not necessarily cause distorsion (e.g. node v_3 in figure 4.2). From this example it becomes clear, the distorsion caused by a node, with respect to its *successors* and *predecessors*, in the general case is :

- 0 in the best case
- $(|V|-1)(T_{max}-2)$ in the worst case (see figure 4.4), i.e. all nodes are forced to start in the latest cycle of their start-interval, if v_1 is started in cycle 1.

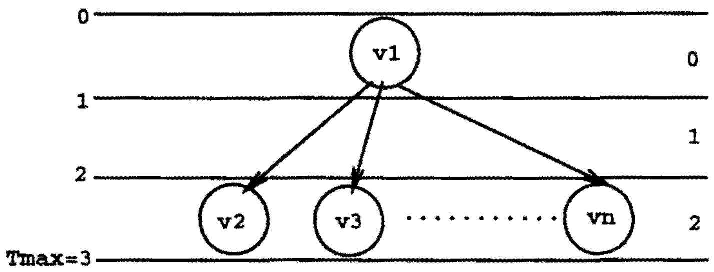

Figure 4.4: Worst-case situation for distorsion.

So far, only the distorsion with respect to predecessors and successors has been discussed, yet there exists another type of distorsion : two nodes, that can be mapped onto the same module in the same cycle, are competitors. Evidently, if one of these nodes is started in such a cycle, it might cause distorsion with respect to the other node (i.e. the competitor). In figure 4.5 a situation is drawn, in which two competitors compete for one module. Suppose, v_1 would start in cycle 1. Then, if there is only one module available to map v_1 and v_2 on, it becomes impossible to start v_2 in either cycle of its start-interval. It is said that v_1 distorts all start possibilities of v_2. However, if v_1 is started in cycle 0 or 2, this leaves to v_2 the choice, to start in cycle 2 or 0, respectively. In this case, v_1 distorts just a fraction of all start possibilities of v_2. This type of distorsion can be stated formally as :

$$DIST_{horz}(v_k,c) = \sum_{v_n \in conc(v_k)} \sum_{c_n=c-\delta(v_n)+1}^{c+\delta(v_k)-1} \frac{\mu_{v_n}(c_n)}{|S_{v_n}|} \tag{4.6}$$

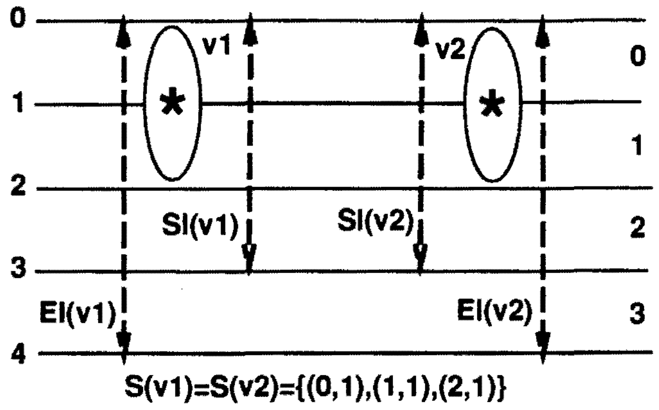

Figure 4.5: Example for horizontal distorsion.

with : $c_n \in C_{v_n}$.

$$conc(v_k) = \{v \in V \mid (EI(v) \cap EI(v_k) \neq \emptyset) \wedge (R_v \cap R_{v_k} \neq \emptyset)\}.$$

Now a measure for the horizontal distorsion is defined, it's necessary to look again at figure 4.5. Notice, that if v_1 would be started in cycle 2, all start-positions of v_2 would be distorted, which is expressed by $DIST_{horz}(v_1, 2) = \frac{3}{3} = 1$; however, starting v_1 in cycle 1 or 3, doesn't affect all start-positions of v_2, which is expressed by $DIST_{horz}(v_1, 1) = DIST_{horz}(v_1, 3) = \frac{2}{3} \approx 0.66$ (Note that interchanging v_1 with v_2 leads to the same results).
In the best case, the horizontal distorsion is 0; in the worst case (e.g. in figure 4.5) it might become impossible to start a node in any cycle of its start-interval. Hence, the greatest horizontal distorsion that can be caused by a node v, equals the number of competitors of the node, i.e. $\mid conc(v) \mid$.

4.3 The fuzzy algorithm

In section 4.2.1, the fuzzy sets applied to scheduling were introduced. Given these definitions, in section 4.2.2 three measures were introduced to investigate the influence of a node, with respect to its successors, predecessors and competitors. Combining the results of section 4.2.1 and section 4.2.2 enables us to design a schedule algorithm, based on fuzzy set techniques.
In short, for each node v in the data-flow graph, the distorsion with respect to its successors, predecessors and competitors could be determined. Then, if there are defined good measures to map the computed distorsions onto the interval [0,1], to each cycle of a node some kind of *quality-degree* could be assigned (a quality-degree actually is the same as a membership-grade). Now, to each node three *quality-degrees* are associated, the worst *quality-degree* of each cycle of a node should be chosen (note that a chain is as strong as its weakest link !). Remind equation 3.4, here we actually take the fuzzy intersection of three fuzzy sets (see figure 4.6, with cycle 1..n $\in C_v$). The expectation is that repeating these computations (i.e. *iterating* !) a few times, eventually results in *quality-degrees* for each cycle of a node, such, that some algorithm should be

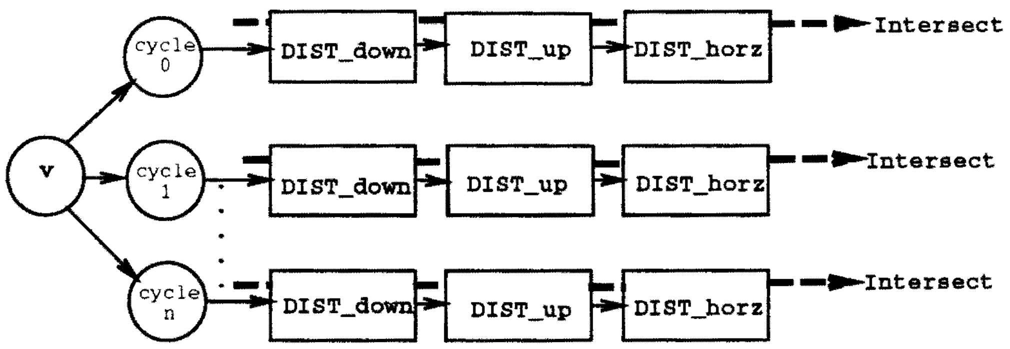

Figure 4.6: Quality-degrees associated to each cycle of a node v.

able to make a feasible schedule. Making a feasible schedule is done by fixing each node of the data-flow graph in the cycle with the highest *quality-degree*, taking into account the available resources. The algorithm can be split in three parts :

- Traverse Top-Down
- Traverse Bottom-Up
- Traverse Horizontal

These three parts will be discussed one by one in the next sections. Finally, an example will be given, that illustrates the operation of the algorithm.

4.3.1 Traverse Top-Down

In this part of the algorithm for all nodes the distorsion is computed with respect to the successors (see definition 4.4). Then to each cycle of a node, a *quality-degree* is assigned. Note that the *quality-degree* assigned to each cycle of a node, actually is the membership-grade of the cycle. Reminding the *quality-distorsion*-plot in figure 4.3, a membership-function should be obtained, that maps the caused distorsion onto the interval [0,1]. The choice of the membership-function is not arbitrary, because remember the aim is to define a function that approximates the shape of the plot in figure 4.3 (see also [Klir88]). As can be seen in figure 4.3, a distorsion equal to zero, results in the highest quality that can be achieved. Hence, a distorsion greater than zero always results in less quality. In other words, *a small distorsion must result in a membership-grade close to 1, a big distorsion must result in a membership-grade close to 0.* The membership-function that shows this behaviour, is :

$$\mu_{down}(DIST_{down}(v,c)) = \frac{1}{1 + \zeta_{down}(DIST_{down}(v,c))^{k_d}} \tag{4.7}$$

with : ζ_{down} some damping parameter, $k_d \in \mathbb{N}$.

The parameter ζ_{down} is used to control the membership-grade for the greatest distorsion present in the data-flow graph, with respect to the highest membership-grade (i.e. 1). E.g. if the maximum distorsion $DIST_{down} = 10$ and given $k_d = 1$, then choosing $\zeta_{down} = 0.9$ results in $\mu_{down}(10) = 0.1$ (note that $\mu_{down}(0) = 1$!). In the final implementation of the algorithm, the user can define ζ_{down} by a so called *Top-Down-Damping-Factor (TDDF)*. This factor represents the fraction of 1, the membership-grade of the cycle with the maximum distorsion $DIST_{down}$ has. Hence, in the case described above, $TDDF = 10$. The parameter k_d is used to control the smoothness of the membership-function. The higher k_d, the steeper the function close to 0 becomes. For $k_d \to \infty$, the

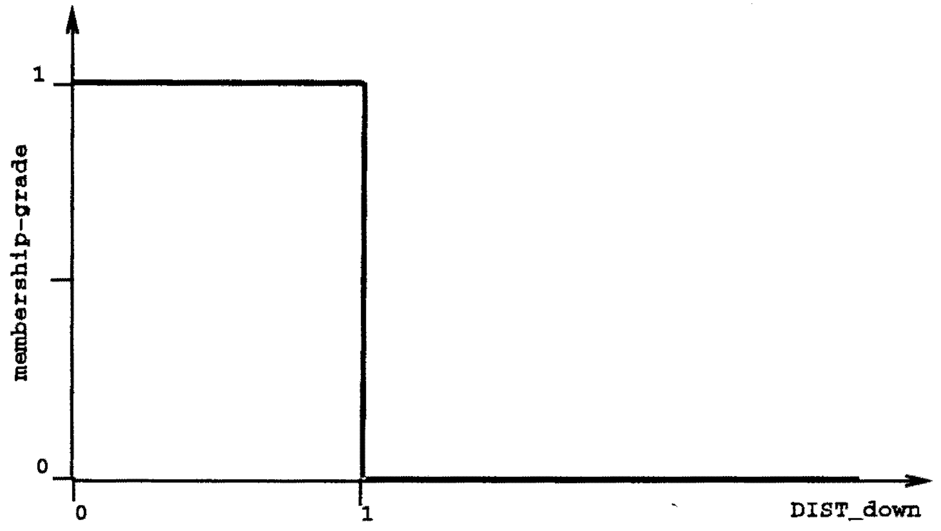

Figure 4.7: μ_{down} for $k_d \to \infty$.

situation as shown in figure 4.7 arises; the result is the membership-function for the set $S = \{x \in \mathbb{R} \mid x \leq 1\}$. Hence, the set S is a *crisp* set !! Finally, the values of ζ_{down} and k_d are chosen empirical and can be set by the user.
Putting all things together, the following pseudo-code can be generated for the *traverse top-down*-procedure :

```
type node = record
              gradeNew : 0..1;
              gradeHorz : 0..1;
              gradeDown : 0..1;
              gradeUp : 0..1
            end;


Traverse Top-Down :
    $v$ : array [| $C_v$ |] of node;
    FOR ALL($v$)
     {
       FOR ALL($c \in C_v$)
           {
              $v$.gradeDown[$c$] := $\mu_{down}(DIST_{down}(v, c))$;
           }
     }
```

In the actual implementation, the recursion present in the formulation for $DIST_{down}$ is used to implement the *traverse top-down* as a recursive algorithm : then each node has to be visited only once, and on top of that, the distorsion for a node with respect to a direct successor is the only computation that has to be done (because the distorsion the regarded successor causes with respect to its own successors, can be stored in the successor itself !). Note, that for simplicity, the algorithm above does not show recursion explicitly.

4.3.2 Traverse Bottom-Up

The procedure for traversing bottom-up differs only slightly from the traverse top-down-procedure. A node also affects its predecessors, if it would be started in a certain cycle : the sooner a node would be started, the greater the distorsion of its predecessors would be. A membership-function, similar to definition 4.7 can be defined for this part :

$$\mu_{up}(DIST_{up}(v,c)) = \frac{1}{1+\zeta_{up}(DIST_{up}(v,c))^{k_u}} \tag{4.8}$$

with : ζ_{up} some damping parameter, $k_u \in \mathbb{N}$.

Similar to definition 4.7 the parameters ζ_{up} and k_u can be set by the user also (as with ζ_{down}, ζ_{up} is set with the *Bottom-Up-Damping-Factor (BUDF)*). Generally, the parameters k_u and k_d are chosen equal; the parameters ζ_{up} and ζ_{down} are chosen such, that $\mu_{down}(DIST_{down}max) = \mu_{up}(DIST_{up}max)$, with $k_d = k_u$ and $DIST_{down}max$ and $DIST_{up}max$ the maximum caused distorsions in the data-flow graph (hence, in this case $TDDF = BUDF$!). This choice is not arbitrary : if the membership-grades for the distorsions $DIST_{down}$ and $DIST_{up}$ are compared, this comparison should be fair. E.g. if for a certain cycle $DIST_{down}$ is half the maximum-$DIST_{down}$ and $DIST_{up}$ is half the maximum-$DIST_{up}$ also, the resulting membership-grades should be equal, otherwise, there would be a preference for starting a node as soon as possible ($\zeta_{up} \gg \zeta_{down}$), or starting a node as late as possible ($\zeta_{up} \ll \zeta_{down}$).

The pseudo-code for the *traverse bottom-up*-procedure then becomes :

```
Traverse Bottom-Up :
    $v$ : node;
    FOR ALL($v$)
     {
      FOR ALL($c \in C_v$)
       {
       $v$.gradeUp[$c$] := $\mu_{up}(DIST_{up}(v,c))$;
       }
     }
```

Similar to the *traverse top-down*-algorithm, the *traverse bottom-up*-algorithm uses the recursion in the definition for $DIST_{up}$ in the actual implementation.
After performing *traverse top-down* and *traverse bottom-up* to each cycle $c \in C_v$ two membership-grades have been assigned. Assuming that each cycle $c \in C_v$ already had

an ancient membership-grade, the new computed grades can be compared with the old ones. Comparing the two values, is done, by taking the fuzzy intersection (see equation 3.4). The computation-order can be stated formally as :

$$\mu^k = MIN([\mu_{down}(DIST_{down}(v,c)]^N, [\mu_{up}(DIST_{up}(v,c))]^N, [\mu^{(k-1)}]^N) \qquad (4.9)$$

with : μ^k the new computed membership-grade, $\mu^{(k-1)}$ the old membership-grade, $\mu_{down}(..)$ and $\mu_{up}(..)$ the computed membership-grades. Note that all computation-values are normalized (on the greatest membership-grade in C_v, as defined in equation 3.5), denoted by $[..]^N$. Now the pseudo-code for the normalization and intersection becomes :

```
NormalizeAndCompare :
    $v$ : array [| $C_v$ |] of node;
    FOR ALL($v$)
     {
      NORM($v$.gradeDown);
      NORM($v$.gradeUp);
      $v$.gradeNew[c] := MIN($v$.gradeDown[c],$v$.gradeUp,$v$.gradeNew[c]);
      NORM($v$.gradeNew);
     }
```

with :

- NORM(..) a normalization-function as defined in equation 3.5
- MIN(..) the (fuzzy) intersection as defined in equation 3.4

It is necessary to normalize the membership-grades, in order to be able to make a fair comparison (=intersection) between two values, e.g. :

Given a node v, with $S_v^{orig} = \{(0,1),(1,0.8),(2,0.6)\}$

Suppose the results for μ_{down} and μ_{up} are :

cycle 0 :	$\mu_{down} = 0.48$	$\mu_{up} = 1$
cycle 1 :	$\mu_{down} = 0.48$	$\mu_{up} = 0.9$
cycle 2 :	$\mu_{down} = 0.6$	$\mu_{up} = 1$

If the intersection of S_v^{orig} with the new μ's is taken, without normalization, the result would be : $S_v^{new} = \{(0,0.48),(1,0.48),(2,0.6)\}$. This means that cycle 2 would have the highest membership-grade. However, if the μ's are normalized first, the result for the intersection becomes : $S_v^{new} = \{(0,0.8),(1,0.8),(2,0.6)\}$, that is, cycle 0 and 1 have a higher membership-grade than cycle 2. Note, this is what was to be achieved, because the normalized membership-grades μ_{down} : cycle 0 : $\mu_{down} = 0.8$, cycle 1 : $\mu_{down} = 0.8$ and cycle 2 : $\mu_{down} = 1$, show, that compared with S_v^{orig}, cycle 1 has an equal membership-grade. This means that there should be no distintion, and thus it is necesarry to normalize.

4.3.3 Traverse Horizontal

The third part of the fuzzy algorithm inspects the quality of the decision to start a node in a certain cylce, with respect to all competitors of this node. In figure 4.5 an example in which several nodes are competing for a certain module, is shown. When a pronouncement of the quality of the decision, to start a node in a certain cycle, must be done, it's to be expected from the foregoing, that (see figure 4.5) for v_1 cycles 1 and 3 would be prefered to cycle 2. So, the greater the distorsion, the worse the quality of the decision, to really start a node in the given cycle. In other words : a big distorsion results in a membership-grade close to 0, a small distorsion results in a membership-grade close to 1. This can be expressed by the following membership-function :

$$\mu_{horz}(DIST_{horz}(v,c)) = \begin{cases} 1 & \text{if } \frac{DIST_{horz}(v,c)-(M-1)}{M} < 0 \\ \frac{1}{1+\zeta_{horz}(\frac{DIST_{horz}(v,c)-(M-1)}{M})^{k_h}} & \text{if } \frac{DIST_{horz}(v,c)-(M-1)}{M} \geq 0 \end{cases} \tag{4.10}$$

with :

- M the number of modules $m \in L$, where v_1 and v_2 can be mapped onto.
- ζ_{horz} some damping parameter, $k_h \in \mathbb{N}$.

The parameters ζ_{horz} and k_h are used in the same way as the according parameters in the functions μ_{down} and μ_{up}. Empiric, it was found out, that choosing $k_h \approx 3 \cdot k_d$ and $\zeta_{horz} \approx \frac{1}{2} \cdot BUDF$ leads to good results.

As can be seen from definition 4.10, the number of modules M is also taken into account. If the number of modules is two (with respect to figure 4.5), it doesn't matter in which cycle the two nodes are started, because each node has the disposal of a module. However, if there is only one module available onto which v_1 and v_2 can be mapped, starting either v_1 or v_2 in cycle 2, leads to problems. In this case it becomes impossible to start the remaining node, once the first node is started in cycle 2. For the *traverse horizontal*-algorithm, the following pseudo-code can be generated :

```
TraverseHorizontal :
    $v$ : array [| $C_v$ |] of node;
    FOR ALL($v$)
     {
      FOR ALL($c \in C_v$)
       {
        $v$.gradeHorz[$c$] := $\mu_{horz}(DIST_{horz}(v,c))$;
       }
     }
```

It's worth mentioning, that the function $DIST_{horz}$ within the *traverse horizontal*-procedure, calculates with the values stored in *gradeNew* after running *traverse top-down* and *traverse bottom-up*.

4.3.4 An example

In this section, an example is given to illustrate the operation of the fuzzy algorithm as discussed in the previous sections. In figure 4.8 a simple data-flow-graph is shown.

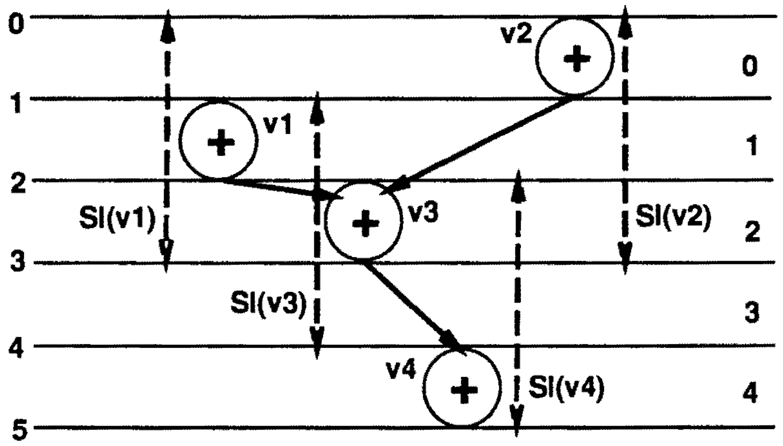

Figure 4.8: Simple data-flow-graph.

With respect to figure 4.8, the following assumptions are made :

- For all $v \in V$, the membership-grades of all elements in S_v are initially set to one.
- The time-constraint $T_{max} = 5$ is given
- The resource-constraint is 1 adder
- $\delta(v_1, adder) = \delta(v_2, adder) = \delta(v_3, adder) = \delta(v_4, adder) = 1$
- $\zeta_{down} = \zeta_{up} = 1$; $\zeta_{horz} = 10$

Below, the results of the different steps are shown in tabular form :

Results top-down

cycle $\rightarrow$	0	1	2	3	4	
v_1	0	2	4	-	-	$DIST_{down}$
	1	0.2	0.06	-	-	$\mu_{down}(DIST)$
v_2	0	2	4	-	-	$DIST_{down}$
	1	0.2	0.06	-	-	$\mu_{down}(DIST)$
v_3	-	0	1	2	-	$DIST_{down}$
	-	1	0.5	0.2	-	$\mu_{down}(DIST)$
v_4	-	-	0	0	0	$DIST_{down}$
	-	-	1	1	1	$\mu_{down}(DIST)$

Results bottom-up

cycle →	0	1	2	3	4	
v_1	0	0	0	-	-	$DIST_{up}$
	1	1	1	-	-	$\mu_{up}(DIST)$
v_2	0	0	0	-	-	$DIST_{up}$
	1	1	1	-	-	$\mu_{up}(DIST)$
v_3	-	4	2	0	-	$DIST_{up}$
	-	0.06	0.2	1	-	$\mu_{up}(DIST)$
v_4	-	-	6	3	0	$DIST_{up}$
	-	-	0.03	0.1	1	$\mu_{up}(DIST)$

Results after intersection and normalization

cycle →	0	1	2	3	4	
v_1	1	0.2	0.06	-	-	$Inters$
	1	0.2	0.06	-	-	$Normalize$
v_2	1	0.2	0.06	-	-	$Inters$
	1	0.2	0.06	-	-	$Normalize$
v_3	-	0.06	0.2	0.2	-	$Inters$
	-	0.3	1	1	-	$Normalize$
v_4	-	-	0.03	0.1	1	$Inters$
	-	-	0.03	0.1	1	$Normalize$

Results horizontal

cycle →	0	1	2	3	4	
v_1	0.79	0.29	0.51	-	-	$DIST_{horz}$
	0.29	0.99	0.85	-	-	$\mu_{horz}(DIST)$
	0.29	1	0.86	-	-	$Normalize$
v_2	0.79	0.29	0.51	-	-	$DIST_{horz}$
	0.29	0.99	0.85	-	-	$\mu_{horz}(DIST)$
	0.29	1	0.86	-	-	$Normalize$
v_3	-	0.32	0.12	0.09	-	$DIST_{horz}$
	-	0.98	0.99	0.99	-	$\mu_{horz}(DIST)$
	-	0.99	1	1	-	$Normalize$
v_4	-	-	0.53	0.43	0	$DIST_{horz}$
	-	-	0.82	0.94	1	$\mu_{horz}(DIST)$
	-	-	0.82	0.94	1	$Normalize$

Results after intersection and normalization

cycle $\rightarrow$	0	1	2	3	4	
v_1	0.29	0.2	0.06	-	-	*Inters*
	1	0.67	0.21	-	-	*Normalize*
v_2	0.29	0.2	0.06	-	-	*Inters*
	1	0.67	0.21	-	-	*Normalize*
v_3	-	0.3	1	1	-	*Inters*
	-	0.3	1	1	-	*Normalize*
v_4	-	-	0.03	0.1	1	*Inters*
	-	-	0.03	0.1	1	*Normalize*

Assuming, after the last procedure (i.e. traverse horizontal and normalization), a *greedy-feasibility*-scheduler is used to assign to each node the real start- and end-time, the results presented in the last table commit with the wished schedule. Note that in chapter 5, the characteristics of the *greedy-feasibility*-scheduler are outlined.

4.4 Initialization and Convergence of the algorithm

In the example shown in section 4.3.4, the membership-grades of the elements in the fuzzy sets are set to 1 initially. Choosing all membership-grades 1, implicates that in a fuzzy set C_v, there is no preference for a certain cycle c_i to a cycle c_j in this set. The algorithm must then determine whether or not there exists a preference-order among the elements of a fuzzy set C_v.

Bearing this in mind, the question could arise what the consequences for the results would be if the membership-grades are initialized with a value different from 1. Two cases must be distinguished :

- *Initialization with the same value* : in this case all elements of a fuzzy set C_v are initialized with the same value (different from 0 !!). Note that this case is similar to the one outlined above (i.e. initialization with 1), because same values implicate no preference-order. Hence, the results will not differ from the results in case of initialization with 1.

- *Random initialization* : in this case the elements of a fuzzy set C_v are all initialized with a different, *random* value. This means that initially there exists a certain preference-order among the elements in this fuzzy set. An interesting question is whether the algorithm will converge to the same results as with initialization with 1, given these *random* values. In other words, do the results of the algorithm depend on the initial values ? Note, that if the algorithm could overcome a bad initialization, it means that it could overcome also a bad choice during run-time. As will be shown in chapter 5, the algorithm converges indeed given a random initialization.

4.5 Complexity of the algorithm

In this section a derivation of the complexity of the algorithm is presented. Note that the *top-down-* and *bottom-up*-traversals are implemented as *depth-first-search*-algorithms. Moreover, all nodes in the data-flow-graph are linked, that is, there exists a link between two nodes if they are concurrents in a certain cycle (see definition 4.6). The concurrent-nodes are linked in a doubly-linked chain.
The complexity of the fuzzy algorithm can be derived now as follows :
The complexity of a depth-first-search-algorithm is equal to $|V| + |E|$. On top of that, for each node being visited, at most $|V| - 1$ direct successor-nodes must be visited, and each direct successor-node can be shifted up or down at most $T_{max} - 1$ cycles (so for the distorsion, at most $T_{max} - 1$ computations must be made) . All together, for the depth-first-search-procedures this results in (note that T_{max} is the time-constraint given) :

$$|V| \cdot (|V| - 1) \cdot T_{max} + |E| \tag{4.11}$$

For the *horizontal*-traversal, at worst a chain of length $|V|$ must be inspected (i.e. all nodes are linked in one and the same link). In the implementation of the *horizontal*-traversal, the chain is inspected from head to tail and vice versa. In formula, this looks like :

$$2 \cdot |V| \cdot T_{max} \tag{4.12}$$

Combining (4.11) and (4.12), and noting that (4.11) must be counted twice (traverse top-down and bottom-up), the total complexity for one iteration becomes :

$$(2 + |V|) \cdot |V| \cdot T_{max} + |E| \tag{4.13}$$

And thus the order of the algorithm becomes :

$$\mathcal{O}(|V|^2 \cdot T_{max} + |E|) \tag{4.14}$$

However, some conditions can be relaxed a little bit. For instance, it's not unrealistic to assume that in a data-flow-graph on average each node has at most two outgoing and two incoming edges. This assumption also implicates that each node on average, has at most two direct successors or predecessors. Hence, in equation 4.11 the factor $(|V| - 1)$ can be substituted by 2. Now the complexity of the fuzzy algorithm becomes :

$$4 \cdot |V| \cdot T_{max} + |E| \tag{4.15}$$

and the order of the algorithm becomes :

$$\mathcal{O}(|V| \cdot T_{max}) \tag{4.16}$$

Chapter 5

Results

5.1 Introduction

This chapter dicusses the results of the fuzzy algorithm. Note that in appendix A some test-graphs are depicted and that appendix B contains the actual results of the algorithm applied to the corresponding graphs in appendix A. Furthermore, appendix D shows for each graph of appendix B the CPU-time used as a function of the number of iterations and appendix E shows the results of the algorithm if the membership-grades are initialized randomly. For illustration, in section 5.2 an example is presented, to show the problems an ordinary list-scheduler might encounter due to its greed, and how the fuzzy algorithm has solved these problems. Finally, in section 5.3, a short evaluation of the results as depicted in appendix A and B is given.

5.2 An example

In figure 5.1, the data-flow graph to be scheduled is shown. It is assumed that $T_{max} = 7$, and that the following resources are available : 1 multiplier, 2 adders and 1 subtractor. The resources are selected from the library :
$L = \{< multiplier : * >, < adder : + >, < subtractor : - >\}$.
Furthermore, $\delta(+) = \delta(-) = 1$ cycle, and $\delta(*) = 2$ cycles. Given these constraints, it becomes clear that only one multiplication at a time can be executed. The computed asap-alap-values for the graph, result in the following start-intervals : $SI(v_4) = SI(v_5) = [2,5]$. For the subtractions also, only one at a time can be executed (the corresponding start-intervals are : $SI(v_6) = SI(v_7) = [4,7]$).

Scheduling the given example with some kind of greedy scheduler (e.g. a list-scheduler), could possibly result in a non-feasible schedule : due to its greedy kind, these schedulers would make bad decisions at crucial moments. For instance, in this example, a bad choice for the placement of a multiplier, results in a non-feasible schedule : if node v_4 or node v_5 is started in cycle 3, due to the constraints it won't be possible anymore, to make a feasible schedule. So, to obtain a feasible schedule, node v_4 or v_5 must be started in cycle 2, forcing the remaining node to be started in cycle 4.

However, once again, a bad choice of the placement of the multipliers can result in a non-feasible schedule : starting node v_5 in cycle 2, and thus starting node v_4 in cycle 4, forces both nodes v_6 and v_7 to start in cycle 6. Hence, this results in a non-feasible schedule, because there is only one subtractor available to execute v_6 and v_7.

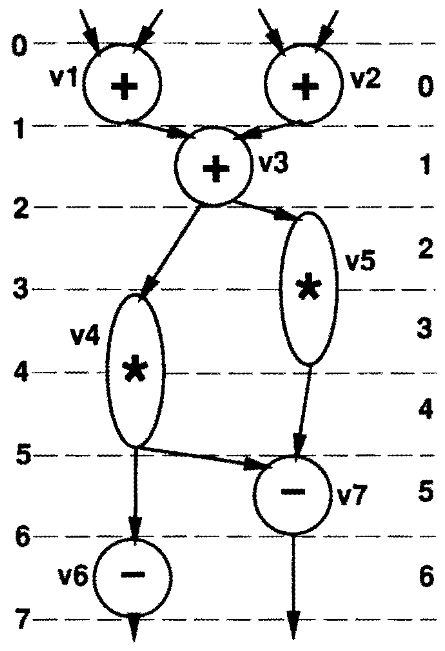

Figure 5.1: Data-flow graph fuzex6 to be scheduled.

All together, this means that eventually only the choice to start v_4 in cycle 2 and v_5 in cycle 4, can result in a feasible schedule. This choice then also fixes node v_7 in cycle 6, and leaves the choice to node v_6 to start in cycle 4 or 5.

From the discussion above, it is to be expected, that for each node v, the cycle wherein the node must start (to achieve a feasible schedule), will have the highest membership-grade of the set C_v. The results of the fuzzy algorithm (presented in appendix B) correspond to these expectations, that is, the cycle in which an operation should start has the highest membership-grade.

5.3 Evaluation

Evaluating the results presented in appendix B, the following remarks can be made :

- Using the computation-schemes of equation 4.9 and figure 4.6, good results can be achieved with only a small number of iterations (2 to 10 iterations), however, increasing the number of iterations (e.g. 100 iterations) does not alter (or improve) the results significantly
- The algorithm is fast : given the fuzex6-graph of appendix A, the algorithm needs 0.01 seconds CPU-time for 2 iterations, while increasing the number of iterations to 100 results in 0.4 seconds CPU-time. Finally, 200 iterations for this graph take 0.82 seconds CPU-time. As can be seen also from the other tables in appendix D, the CPU-time needed increases linear with the number of iterations
- The results presented in appendix E show that if the membership-grades are initialized at random, the algorithm still produces good results. However, the

initialized membership-grades in a fuzzy set must be of the same order. As can be seen from the second test in appendix E, initializing some membership-grades with a factor 10 smaller (or bigger) may eventually lead to the wrong results. In this case the nodes v_6 and v_7 have changed positions, that is, v_6 is assigned the highest membership-grade in a cycle where it should definitely not be started !!! Note the same holds for node v_7. As outlined in section 4.4, the algorithm converges indeed. This implicates that if in a certain iteration-step an unfortunate decision is made, the algorithm will be able to overcome this choice in later iteration-steps, thus trying to achieve the best solution

- The (user-defined) parameters of the algorithm affect the schedule-results, i.e. different membership-grades for the cycles are computed. Therefore (although it hasn't showed yet) it could be possible that choosing the wrong parameters results in less meaningful (and hence less useful) membership-grades of the cycles of a node. As a rule of thumb, choosing $BUDF = TDDF$, $\zeta_{horz} \approx \frac{1}{2} \cdot BUDF$, $k_d = k_u$ and $k_h \approx 3 \cdot k_d$, give the best results.

5.4 Greedy-feasibility scheduler

The results as presented in appendix B, only indicate the quality of the decision to start a node in a certain cycle. However, the nodes are not fixed yet ! Therefore, an other algorithm must perform this actual scheduling (i.e. fixing nodes in cycles); this can be done by various types of schedulers (e.g. list-schedulers, force-directed schedulers). A scheduler, offering good results in combination with the results of the fuzzy algorithm, is a greedy-feasibility scheduler. This scheduler traverses top-down, and fixes a node in a cycle, when :

- All its predecessors are executed
- For the regarded cycle, the node has the highest membership-grade with respect to all its competitors
- There is at least one module available that can execute the node

After fixing a node in a cycle, the membership-grades for all successors (and the successors of these successors, etc.) must be updated !! In appendix C, the results of this scheduler applied to the results of appendix B are presented.

5.5 Recommandations for future work

In this section the short-comings of the fuzzy algorithm are discussed, and some recommandations for future work are done.
In short, the short-comings are :

- The algorithm cannot handle non-feasibility in an unambigous way. That means, that a greedy-feasibility scheduler as discussed in section 5.4, can only make a schedule if the data-flow graph is feasible. Otherwise, the greedy-feasibility scheduler would terminate, resulting in no schedule. In this case it's not clear, whether the scheduler made a mistake or that the graph is non-feasible (due to the time- and resource-constraints given).

- A bad choice of the parameters ζ_{down}, ζ_{up}, ζ_{horz} may result in results that are not useful, or when they *are* used, would lead to an non-feasible schedule. However, it is not easy to determine the values for these parameters, that would offer the best result. So far, the parameters are estimated on an empirical basis. Therefore it would be desirable to search for characteristics of data-flow graphs, where values of the parameters could be derived from.
- The algorithm is iterative, so it is to be expected that the solution approaches a feasible (and thus correct) schedule, as the number of iterations increases. Unfortunately, the algorithm is not yet provided with a tool to measure the quality of the schedule, at any moment. This means, that using just a few iterations could possibly result in a bad schedule, whereas a few more iterations would have resulted in a much better schedule. Possibly genetic algorithms can be useful to determine the quality of the solution. Another possiblity could be the introduction of the *fuzzy entropy* (FE) for a membership-grade m of an element [Klir88] :

  $FE(m) = -m \cdot log(m) - (1 - m) \cdot log(1 - m)$

  Actually, the *goal* of the fuzzy algorithm is to assign a membership-grade to each $c \in C_v$ of a node v. In the best case, a membership-grade 0 is assigned to a cycle, if a node *is not* allowed to be executed (or executing) in that cycle, a membership-grade 1 is assigned to a cycle if the node *is* allowed to be executed (or executing) in that cycle. In this case $\sum_{c \in C_v} FE(\mu_v(c)) = 0$, because for any $c \in C_v$ for each $v \in V, \mu_v(c) = 0$ or $\mu_v(c) = 1$. So, the closer $FE(m)$ is to 0, the closer the membership-grades are to 0 or 1, and thus the better the schedule.

  A possibility to move the membership-grades to 0 or 1, could be the introduction of a trigger-function as discussed in [Zad68]. In short, this function pulls all membership-grades below a certain "threshold" to 0, and all membership-grades above this "threshold" to 1. The threshold-value depends on the membership-grades of all the elements of a fuzzy set.
- The algorithm cannot handle complex data-flow graphs : graphs containing pipelined operations and operations that can be mapped on different types of modules are not allowed. Therefore the algorithm must be extended with features to be able to handle these constructs, in order to be useful as a real design-tool.

Chapter 6

Conclusions

This report is concerned with the following problem :
Given a data-flow graph, a time constraint and a resource constraint, find a feasible schedule. In chapter 4, an heuristic schedule-algorithm, based on fuzzy set techniques has been discussed.

The advantages of this algorithm are :

- The algorithm offers good solutions and has proved to be able to handle situations wherein other schedulers (e.g. an ordinary list-scheduler) would probably fail.
- The algorithm is fast : on average the algorithm is linear with the number of nodes (in the data-flow graph) and the time-constraint.
- The results of the algorithm can be used by other schedulers, for instance as initialization.
- The algorithm converges to a solution, indepent from the initial membership-grades.

A disadvantage of the algorithm is that the produced results of the algorithm strongly depend on the choice of some parameters

Finally, working out the recommandations discussed in section 5.5, will almost certainly overcome the short-comings of the fuzzy algorithm.

Bibliography

[Denyer90] **A New Approach To Pipeline Optimisation**
Denyer P.B., Mallon D.J.
EDAC 1990, pp.83-88

[Eijndhoven91] **The ASCIS data flow graph: semantics and textual format**
Eijndhoven J.T.J. van, Jong G.C. and Stok L.
EUT report 91-E-251, April 1991

[Feli90] **Goal-oriented control of VLSI-Design processes based on fuzzy sets**
Felix R.
Proc. of the 20th International Symposium on Multiple-Valued Logic, Charlotte, North Carolina, May 1990, pp.386-393

[Feli91] **Goal-oriented high-level synthesis based on a fuzzy decision theory**
Felix R., Powswig J.
Clear Applications of Fuzzy Logic, IEEE Symposium, Delft University of Technology (The Netherlands), 17 October, 1991, pp.130-139

[Gusev75] **Fuzzy Sets theory and applications**
Gusev L.A. and Smirnova I.M.
Automation and Remote Control (USA), Vol.34 (May 1973), No.5,pp.739-755

[Heijligers91] **Time constrainted scheduling for high-level synthesis**
Heijligers M.J.M.
Eindhoven University of Technology, May 1991, Master Thesis

[Klir88] **Fuzzy Sets, Uncertainty and Information**
Klir G.J. and Folger T.A.
New Jersey, Prentice-Hall, 1988

[Mowch91] **Bottom Up Synthesis based on fuzzy schedules**
Ly, T.A. and Mowchenko, J.T.
Proc. of the 28th ACM/IEEE Design Automation Conference, June, 1991, pp.674-679

[Negoi81] **Fuzzy Systems**
Negoita C.V.
Kent, Abacus Press, 1981

[Paulin89] **Scheduling and Binding algorithms for high-level synthesis**
Paulin P.G.
Proc. of the 26th ACM/IEEE Design Automation Conference,January,1989, pp.1-6

[Prad79] **Using fuzzy set theory in a scheduling problem : a case study**
Prade, H
Fuzzy Sets and Systems, Vol. 1 (1979), No. 2, pp.129-149

[Zad65] **Fuzzy Sets**
Zadeh L.A.
Information and Control (USA), Vol.8 (1965), No.8, pp.338-353

[Zad68] **Communication : Fuzzy Algorithms**
Zadeh L.A.
Information and Control (USA), Vol.12 (1968), No.2, pp.94-102

Appendix A

Test-graphs

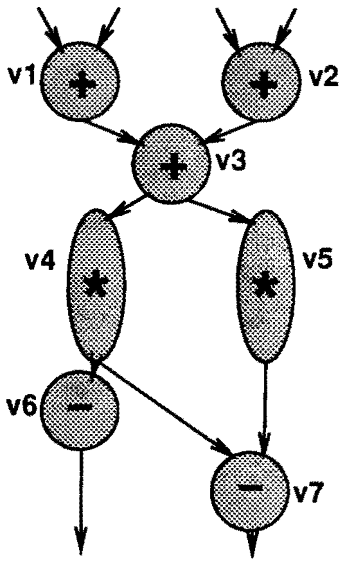

Figure A.1: Fuzex6-graph.

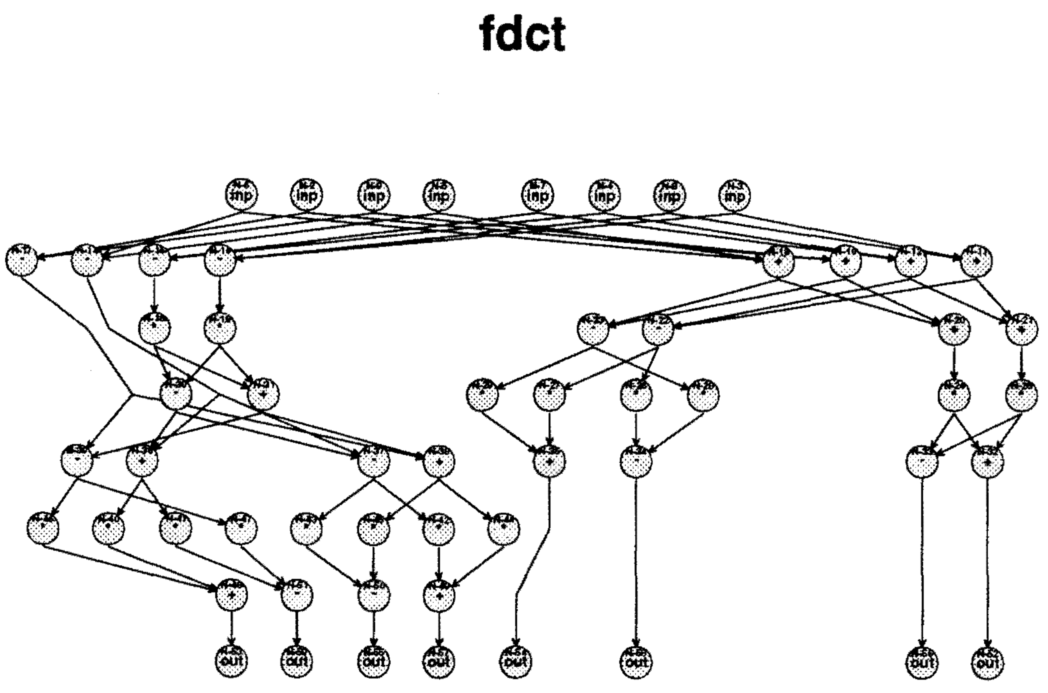

Figure A.2: Fdct-graph from [Denyer90].

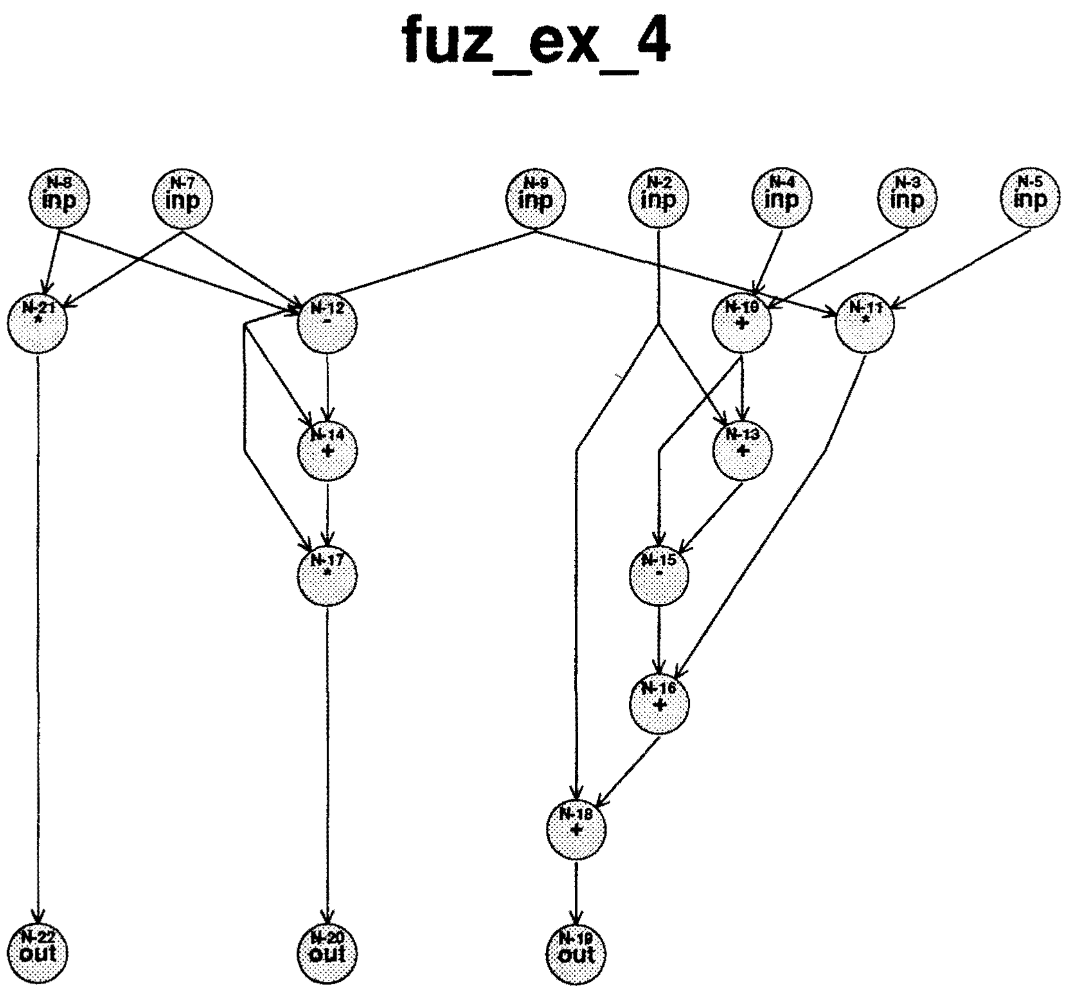

Figure A.3: Fuzex4-graph.

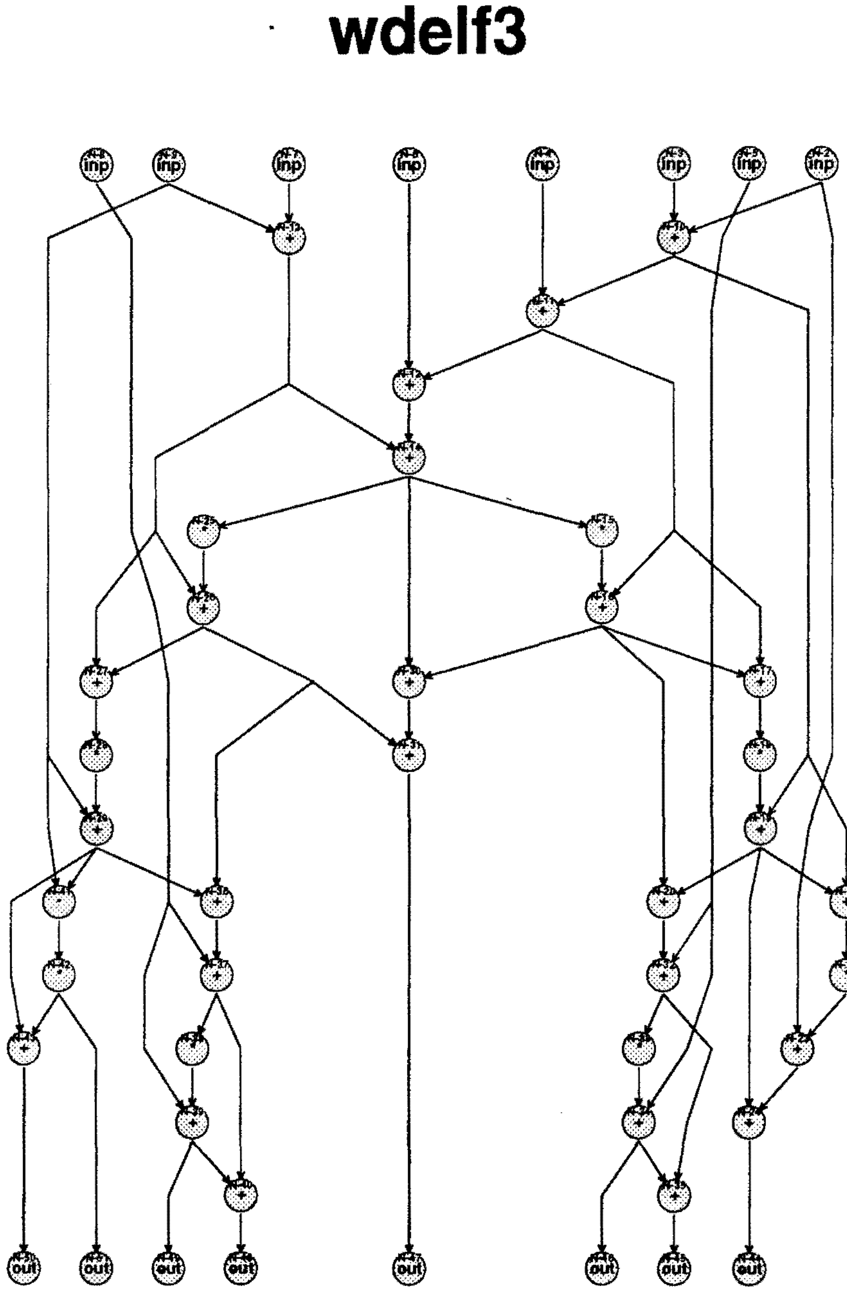

Figure A.4: Wdelf3-graph.

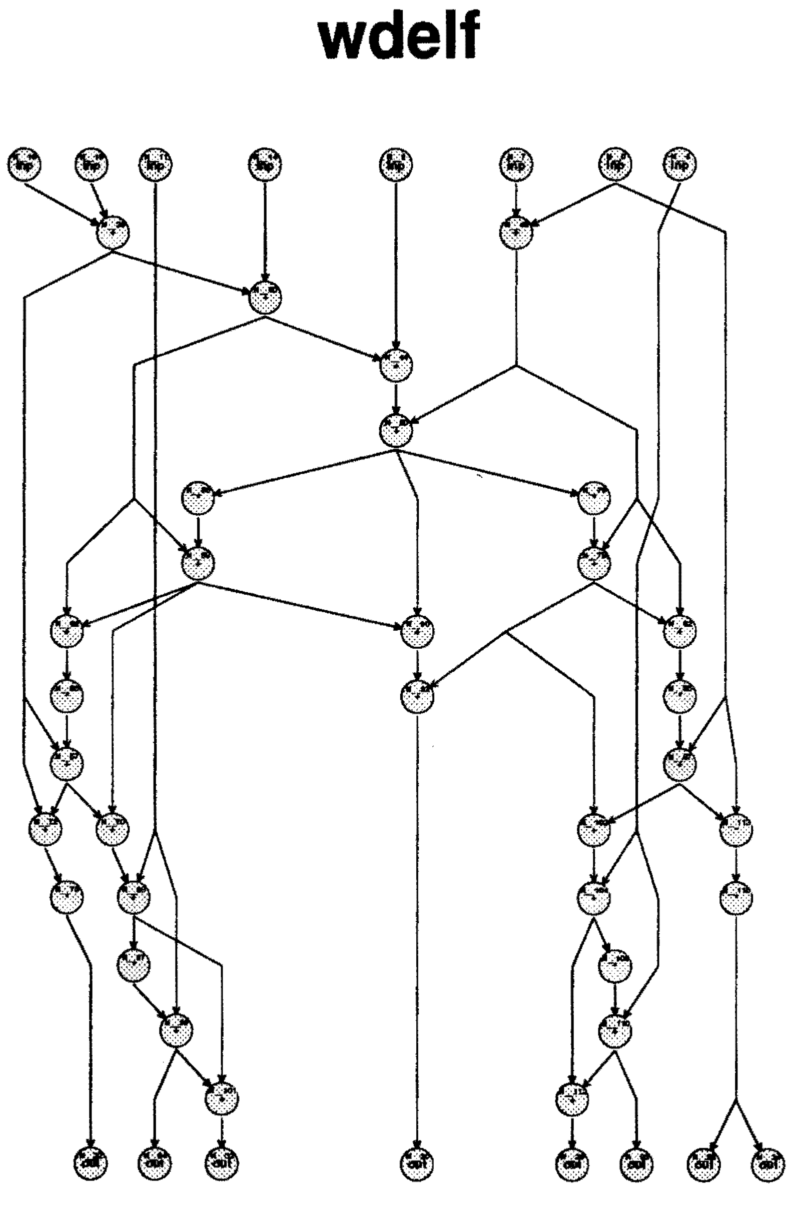

Figure A.5: Wdelf2-graph. Note the wdelf2-graph is a slight modification of the wdelf3-graph : the nodes N-23, N-24 and N-43 are removed from the wdelf3-graph.

Appendix B

Schedule results

In this appendix the results of the fuzzy algorithm are presented; the according data-flow graphs are shown in appendix A. In the tables, an entry denotes the membership-grade m for the regarded cycle $c \in C_v$ of node v.

Results fuzex6-graph :
Library $L = \{< multiplier : * >, < adder : + >, < subtractor : - >\}$.
Resources : 1 multiplier, 2 adders, 1 subtractor.
Delay of operations : $\delta(adder) = \delta(subtractor) = 1, \delta(multiplier) = 2$.
Time-constraint : $T_{max} = 7$.

Number of iterations = 2
$BUDF = TDDF = 10,\ \zeta_{horz} = 100,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.31	0.10	0	0	0	0
$v_2(+)$	1.00	0.31	0.10	0	0	0	0
$v_3(+)$	0	1.00	0.62	0.22	0	0	0
$v_4(*)$	0	0	1.00	0.04	0.42	0	0
$v_5(*)$	0	0	0.15	0.02	1.00	0	0
$v_6(-)$	0	0	0	0	1.00	0.74	0.17
$v_7(-)$	0	0	0	0	0.45	0.10	1.00

Total run-time : 0.01

Number of iterations = 4
$BUDF = TDDF = 10,\ \zeta_{horz} = 100,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.31	0.10	0	0	0	0
$v_2(+)$	1.00	0.31	0.10	0	0	0	0
$v_3(+)$	0	1.00	0.65	0.23	0	0	0
$v_4(*)$	0	0	1.00	0.01	0.01	0	0
$v_5(*)$	0	0	0.01	0.01	1.00	0	0
$v_6(-)$	0	0	0	0	0.91	1.00	0.05
$v_7(-)$	0	0	0	0	0.27	0.10	1.00

Total run-time : 0.01

Number of iterations = 4

$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.00	0.00	0	0	0	0
$v_2(+)$	1.00	0.00	0.00	0	0	0	0
$v_3(+)$	0	1.00	0.02	0.00	0	0	0
$v_4(*)$	0	0	1.00	0.00	0.00	0	0
$v_5(*)$	0	0	0.00	0.00	1.00	0	0
$v_6(-)$	0	0	0	0	0.16	1.00	0.00
$v_7(-)$	0	0	0	0	0.03	0.00	1.00

Total run time: 0.02

Number of iterations = 100

$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.00	0.00	0	0	0	0
$v_2(+)$	1.00	0.00	0.00	0	0	0	0
$v_3(+)$	0	1.00	0.02	0.00	0	0	0
$v_4(*)$	0	0	1.00	0.00	0.00	0	0
$v_5(*)$	0	0	0.00	0.00	1.00	0	0
$v_6(-)$	0	0	0	0	0.17	1.00	0.00
$v_7(-)$	0	0	0	0	0.03	0.00	1.00

Total run time: 0.4

Number of iterations = 200

$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.00	0.00	0	0	0	0
$v_2(+)$	1.00	0.00	0.00	0	0	0	0
$v_3(+)$	0	1.00	0.02	0.00	0	0	0
$v_4(*)$	0	0	1.00	0.00	0.00	0	0
$v_5(*)$	0	0	0.00	0.00	1.00	0	0
$v_6(-)$	0	0	0	0	0.17	1.00	0.00
$v_7(-)$	0	0	0	0	0.03	0.00	1.00

Total run time: 0.82

Results fdct-graph :
Library $L = \{< multiplier : * >, < adder : +, - >\}$.
Resources : 8 multipliers, 4 adders.
Delay of operations : $\delta(adder) = 1, \delta(multiplier) = 2$.
Time-constraint : $T_{max} = 8$.

Number of iterations = 4								
$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$								
$Cycle \rightarrow$	0	1	2	3	4	5	6	7
N-17(-)	0.00	1.00	0.83	1.00	0	0	0	0
N-16(-)	1.00	0	0	0	0	0	0	0
N-15(-)	1.00	0	0	0	0	0	0	0
N-14(-)	0.00	1.00	0.83	1.00	0	0	0	0
N-13(+)	0.77	1.00	0.01	0.00	0	0	0	0
N-12(+)	0.77	1.00	0.01	0.00	0	0	0	0
N-11(+)	0.77	1.00	0.01	0.00	0	0	0	0
N-10(+)	0.77	1.00	0.01	0.00	0	0	0	0
N-23(-)	0	0.06	1.00	0.05	0.02	0	0	0
N-22(-)	0	0.06	1.00	0.05	0.02	0	0	0
N-21(+)	0	0.06	1.00	0.14	0.03	0	0	0
N-20(+)	0	0.06	1.00	0.14	0.03	0	0	0
N-19(*)	0	1.00	0	0	0	0	0	0
N-18(*)	0	1.00	0	0	0	0	0	0
N-29(*)	0	0	0.03	1.00	0.00	0.20	0	0
N-28(*)	0	0	0.03	1.00	0.00	0.20	0	0
N-27(*)	0	0	0.03	1.00	0.00	0.20	0	0
N-26(*)	0	0	0.03	1.00	0.00	0.20	0	0
N-25(*)	0	0	0.04	1.00	0.00	0.08	0	0
N-24(*)	0	0	0.04	1.00	0.00	0.08	0	0
N-31(+)	0	0	0	1.00	0	0	0	0
N-30(-)	0	0	0	1.00	0	0	0	0
N-39(-)	0	0	0	0	1.00	0	0	0
N-38(+)	0	0	0	0	1.00	0	0	0
N-37(-)	0	0	0	0	1.00	0	0	0
N-36(+)	0	0	0	0	1.00	0	0	0
N-35(+)	0	0	0	0	0.01	0.53	1.00	0.16
N-34(-)	0	0	0	0	0.01	0.53	1.00	0.16
N-33(-)	0	0	0	0	0.01	0.34	1.00	0.11
N-32(+)	0	0	0	0	0.01	0.34	1.00	0.11

N-47(*)	0	0	0	0	0	1.00	0	0
N-46(*)	0	0	0	0	0	1.00	0	0
N-45(*)	0	0	0	0	0	1.00	0	0
N-44(*)	0	0	0	0	0	1.00	0	0
N-43(*)	0	0	0	0	0	1.00	0	0
N-42(*)	0	0	0	0	0	1.00	0	0
N-41(*)	0	0	0	0	0	1.00	0	0
N-40(*)	0	0	0	0	0	1.00	0	0
N-51(-)	0	0	0	0	0	0	0	1.00
N-50(-)	0	0	0	0	0	0	0	1.00
N-49(+)	0	0	0	0	0	0	0	1.00
N-48(+)	0	0	0	0	0	0	0	1.00

Total run time : 0.02

Results fuzex4-graph :
Library $L = \{< multiplier : * >, < adder : +, - >\}$.
Resources : 1 multiplier, 2 adders.
Delay of operations : $\delta(adder) = 1, \delta(multiplier) = 2$.
Time-constraint : $T_{max} = 6$.

Number of iterations = 4

$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

$Cycle \rightarrow$	0	1	2	3	4	5
N-10(+) :	1.00	0.00	0	0	0	0
N-11(*) :	1.00	1.00	0.00	0	0	0
N-12(-) :	1.00	0.00	0.00	0	0	0
N-21(*) :	0.03	0.03	1.00	1.00	0.00	0
N-13(+) :	0	1.00	0.01	0	0	0
N-14(+) :	0	0.39	1.00	0.25	0	0
N-15(-) :	0	0	1.00	0.64	0	0
N-17(*) :	0	0	0.00	0.00	1.00	0
N-16(+) :	0	0	0	0.02	1.00	0
N-18(+) :	0	0	0	0	0.00	1.00

Total run time: 0.02

Results wdelf2-graph :
Library $L = \{< multiplier : * >, < adder : +, - >\}$.
Resources : 2 multipliers, 2 adders.
Delay of operations : $\delta(adder) = 1, \delta(multiplier) = 2$.
Time-constraint : $T_{max} = 17$.

Number of iterations = 4
$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
N-36(+)	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-46(+)	1.00	1.00	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-40(+)	0	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-44(+)	0	0	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-50(+)	0	0	0	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0
N-56(*)	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0	0	0	0
N-76(*)	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0	0	0	0
N-58(+)	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0	0
N-78(+)	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0	0
N-62(+)	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0
N-82(+)	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0	0
N-90(+)	0	0	0	0	0	0	0	0.03	1.00	0.29	0.02	0.01	0.00	0.03	0.01	0.01	0
N-65(*)	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0
N-85(*)	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0	0	0
N-92(+)	0	0	0	0	0	0	0	0	0.02	0.09	0.03	0.02	0.00	0.46	1.00	0.03	0.03
N-67(+)	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0
N-87(+)	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0	0
N-70(+)	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0
N-72(+)	0	0	0	0	0	0	0	0	0	0	0	0.02	0.01	1.00	0.45	0	0
N-102(+)	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0	0
N-113(-)	0	0	0	0	0	0	0	0	0	0	0	0.02	0.01	1.00	0.45	0	0
N-75(*)	0	0	0	0	0	0	0	0	0	0	0	0	0.00	0.01	0.00	1.00	0
N-93(+)	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0
N-104(+)	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0	0
N-116(*)	0	0	0	0	0	0	0	0	0	0	0	0	0.00	0.01	0.00	1.00	0
N-97(*)	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0
N-108(*)	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0	0	0
N-99(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0
N-110(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0
N-101(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00
N-112(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00

Total run time: 0.02

As can be seen from this table, the fuzzy algorithm can handle the conflict-situation between the 4 multiplications, i.e. the nodes N-75, N-97, N-108 and N-116. Given the time-constraint $T_{max} = 17$, nodes N-97 and N-108 must be started in cycle 13 because they are on the critical path. This implicates N-75 and N-116 must be started in cycle 15 to achieve a feasible solution.

Results wdelf3-graph :
Library $L = \{< multiplier : * >, < adder : + >, < subtractor : - >\}$.
Resources : 2 multipliers, 2 adders, 1 subtractor.
Delay of operations : $\delta(adder) = 1, \delta(multiplier) = 2, \delta(subtractor) = 1$.
Time-constraint : $T_{max} = 18$.

Number of iterations = 10
$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
N-13(+)	1.00	1.00	1.00	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-10(+)	1.00	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-11(+)	0	1.00	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-12(+)	0	0	1.00	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N-14(+)	0	0	0	1.00	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0
N-25(*)	0	0	0	0	1.00	0.01	0	0	0	0	0	0	0	0	0	0	0	0
N-15(*)	0	0	0	0	1.00	0.01	0	0	0	0	0	0	0	0	0	0	0	0
N-26(+)	0	0	0	0	0	0	1.00	0.02	0	0	0	0	0	0	0	0	0	0
N-16(+)	0	0	0	0	0	0	1.00	0.01	0	0	0	0	0	0	0	0	0	0
N-30(+)	0	0	0	0	0	0	0	0.01	1.00	0.52	0.06	0.04	0.00	0.01	0.05	0.00	0.00	0
N-27(+)	0	0	0	0	0	0	0	1.00	0.03	0	0	0	0	0	0	0	0	0
N-17(+)	0	0	0	0	0	0	0	1.00	0.02	0	0	0	0	0	0	0	0	0
N-31(+)	0	0	0	0	0	0	0	0	0.04	0.16	0.06	0.07	0.00	0.01	1.00	0.12	0.00	0.00
N-28(*)	0	0	0	0	0	0	0	0	1.00	0.06	0	0	0	0	0	0	0	0
N-18(*)	0	0	0	0	0	0	0	0	1.00	0.03	0	0	0	0	0	0	0	0
N-29(+)	0	0	0	0	0	0	0	0	0	0	1.00	0.14	0	0	0	0	0	0
N-19(+)	0	0	0	0	0	0	0	0	0	0	1.00	0.04	0	0	0	0	0	0
N-41(-)	0	0	0	0	0	0	0	0	0	0	0	1.00	0.62	0.23	0.18	0	0	0
N-36(+)	0	0	0	0	0	0	0	0	0	0	0	0.02	1.00	0	0	0	0	0
N-21(+)	0	0	0	0	0	0	0	0	0	0	0	1.00	0.04	0.03	0	0	0	0
N-20(+)	0	0	0	0	0	0	0	0	0	0	0	0.03	1.00	0	0	0	0	0
N-42(*)	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0.01	0.24	0.29	0	0
N-37(+)	0	0	0	0	0	0	0	0	0	0	0	0	0.00	1.00	0	0	0	0
N-32(+)	0	0	0	0	0	0	0	0	0	0	0	0	0.00	1.00	0	0	0	0
N-22(*)	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0.00	0.00	0	0	0
N-38(*)	0	0	0	0	0	0	0	0	0	0	0	0	0	0.01	1.00	0	0	0
N-33(*)	0	0	0	0	0	0	0	0	0	0	0	0	0	0.02	1.00	0	0	0
N-43(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.24	1.00	0.01	0.00
N-23(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	1.00	0.01	0
N-39(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.00	1.00	0
N-34(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.01	1.00	0
N-24(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.00	0.12	0.17
N-40(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.00	1.00
N-35(+)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.00	1.00

Total run time: 0.06

Given the time- and resource-constraints, in cycle 11 the scheduler must make a choice between the plus-operations N-20, N-36 and N-21. If operations N-20 and N-36 are chosen to start in cycle 11, the soonest start-possibility for N-21 becomes cycle 12. However, if the nodes N-37 and N-32 are started in cycle 12, operation N-21 is forced to start in cycle 13. If in this situation N-42 is started in cycle 12, one of the multiplications N-38 or N-33 can start in cycle 13. The remaining one must then start in cycle 14.
Starting N-21 in cycle 13, would make it possible for multiplication N-22 to start in cycle 14, however in cycle 14 the two multiplicators are already occupied by N-33 and N-38. This implicates that multiplication N-22 must start in cycle 15, however, this will violate the time-constraint $T_{max} = 18$, because after N-22, N-23 and N-24 have to be executed. In this case an extra cycle is necessary to obtain a feasible schedule.
Hence a feasible solution that meets the constraints, can be obtained if in cycle 11 operations N-21, N-41 and one of the operations N-20 and N-36 are started. If N-20 is chosen, multiplications N-42 and N-22 can start in cycle 12. In the same cycle,

operations N-32 and N-36 can be executed. Then, in cycle 14 N-33 and N-37 can be started, together with N-43 or N-23. In cycle 15 multiplication N-38 can be started and in the remaining cycles the remaining plus-operators can be started, without violating the time-constraint.
In figure B.1, the operations mentioned above are shown, in the range from cycle 11 to cycle 17. Note that the fuzzy algorithm offers a good solution : the nodes N-41

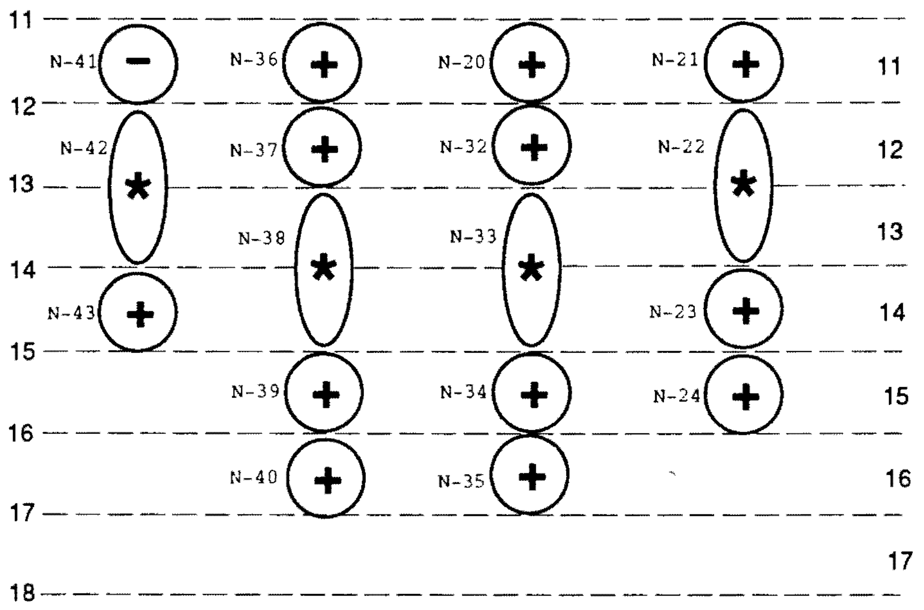

Figure B.1: Detailed view of the Wdelf3-graph.

and N-21 are assigned the highest membership-grade for cycle 11, whereas nodes N-36 and N-20 are assigned the highest membership-grade for cycle 12. It is clear that starting N-21 and N-41 in cycle 11 will eventually produce a feasible solution (note the algorithm leaves the choice to start N-20 or N-36 in cycle 11 !!). It is obvious that these membership-grades can be useful for other schedulers (e.g. list-scheduler, greedy-schedulers) as an initialization, thus enhancing their performance.

Appendix C

Results greedy-feasiblity scheduler

In this appendix, the expected results of a greedy-feasibility scheduler are presented. In the tables, an entry equal to 1, means that the regarded node *must* start in that cycle, an entry equal to 0 means that the node regarded is *not allowed* to start in that cycle.

Results fuzex6-graph :

Number of iterations = 4
$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

$Cycle \rightarrow$	0	1	2	3	4	5	6
$v_1(+)$	1	0	0	0	0	0	0
$v_2(+)$	1	0	0	0	0	0	0
$v_3(+)$	0	1	0	0	0	0	0
$v_4(*)$	0	0	1	0	0	0	0
$v_5(*)$	0	0	0	0	1	0	0
$v_6(-)$	0	0	0	0	1	0	0
$v_7(-)$	0	0	0	0	0	0	1

Results fdct-graph :

Number of iterations = 4
$BUDF = TDDF = 2000,\ \zeta_{horz} = 1000,\ k_d = k_u = 2,\ k_h = 6.$

Cycle →	0	1	2	3	4	5	6	7
N-17(-)	0	1	0	0	0	0	0	0
N-16(-)	1	0	0	0	0	0	0	0
N-15(-)	1	0	0	0	0	0	0	0
N-14(-)	0	1	0	0	0	0	0	0
N-13(+)	1	0	0	0	0	0	0	0
N-12(+)	0	1	0	0	0	0	0	0
N-11(+)	0	1	0	0	0	0	0	0
N-10(+)	1	0	0	0	0	0	0	0
N-23(-)	0	0	1	0	0	0	0	0
N-22(-)	0	0	1	0	0	0	0	0
N-21(+)	0	0	1	0	0	0	0	0
N-20(+)	0	0	1	0	0	0	0	0
N-19(*)	0	1	0	0	0	0	0	0
N-18(*)	0	1	0	0	0	0	0	0
N-29(*)	0	0	0	1	0	0	0	0
N-28(*)	0	0	0	1	0	0	0	0
N-27(*)	0	0	0	1	0	0	0	0
N-26(*)	0	0	0	1	0	0	0	0
N-25(*)	0	0	0	1	0	0	0	0
N-24(*)	0	0	0	1	0	0	0	0
N-31(+)	0	0	0	1	0	0	0	0
N-30(-)	0	0	0	1	0	0	0	0
N-39(-)	0	0	0	0	1	0	0	0
N-38(+)	0	0	0	0	1	0	0	0
N-37(-)	0	0	0	0	1	0	0	0
N-36(+)	0	0	0	0	1	0	0	0
N-35(+)	0	0	0	0	0	1	0	0
N-34(-)	0	0	0	0	0	1	0	0
N-33(-)	0	0	0	0	0	1	0	0
N-32(+)	0	0	0	0	0	1	0	0
N-47(*)	0	0	0	0	0	1	0	0
N-46(*)	0	0	0	0	0	1	0	0
N-45(*)	0	0	0	0	0	1	0	0
N-44(*)	0	0	0	0	0	1	0	0
N-43(*)	0	0	0	0	0	1	0	0
N-42(*)	0	0	0	0	0	1	0	0
N-41(*)	0	0	0	0	0	1	0	0
N-40(*)	0	0	0	0	0	1	0	0
N-51(-)	0	0	0	0	0	0	0	1
N-50(-)	0	0	0	0	0	0	0	1
N-49(+)	0	0	0	0	0	0	0	1
N-48(+)	0	0	0	0	0	0	0	1

Appendix D

Run-time of the algorithm

This appendix contains the run-time results of the algorithm applied to some of the examples shown in appendix A.

Number of iterations	2	4	10	100	200
Used CPU-time (seconds)	0.01	0.02	0.05	0.40	0.82

Fuzex6-graph; $T_{max} = 7$

Number of iterations	2	4	10	100	200
Used CPU-time (seconds)	0.02	0.02	0.05	0.57	1.19

fdct-graph; $T_{max} = 8$

Number of iterations	2	4	10	100	200
Used CPU-time (seconds)	0.01	0.02	0.04	0.43	0.81

Fuzex4-graph; $T_{max} = 6$

Number of iterations	2	4	10	100	200
Used CPU-time (seconds)	0.02	0.03	0.06	0.49	1

Wdelf3-graph; $T_{max} = 16$

Appendix E
Convergence-results

In this appendix the results for a *random* initialization are presented. As test-graph example fuzex6-graph from appendix A is chosen. In the first table the initialization values are shown, followed by the achieved results.

Initialization (random-test 1):

$Cycle \rightarrow$	0	1	2	3	4	5	6
$v_1(+)$	0.2	0.8	0.5	0	0	0	0
$v_2(+)$	0.5	0.7	0.3	0	0	0	0
$v_3(+)$	0	0.1	1	0.4	0	0	0
$v_4(*)$	0	0	0.5	0.9	1	0	0
$v_5(*)$	0	0	1	0.5	0.7	0	0
$v_6(-)$	0	0	0	0	0.2	0.3	0.8
$v_7(-)$	0	0	0	0	0.5	0.8	0.3

The result for this initialization (random-test 1):

$Cycle \rightarrow$	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.00	0.00	0	0	0	0
$v_2(+)$	1.00	0.00	0.00	0	0	0	0
$v_3(+)$	0	1.00	0.00	0.00	0	0	0
$v_4(*)$	0	0	1.00	0.00	0.00	0	0
$v_5(*)$	0	0	0.00	0.00	1.00	0	0
$v_6(-)$	0	0	0	0	0.18	1.00	0.00
$v_7(-)$	0	0	0	0	0.03	0.01	1.00

Number of iterations : 4
TDDF : 2000, **BUDF :** 2000, **HDF :** 1000
kd : 2, **ku :** 2, **kh :** 6
Library : $\{< multiplier : * >, < adder : + >, < subtractor : - >\}$
Resources : 1 multiplier, 2 adders, 1 subtractor.

Initialization (random-test 2):

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	0.02	0.8	0.5	0	0	0	0
$v_2(+)$	0.05	0.7	0.3	0	0	0	0
$v_3(+)$	0	0.01	1	0.4	0	0	0
$v_4(*)$	0	0	0.05	0.4	1	0	0
$v_5(*)$	0	0	1	0.7	0.07	0	0
$v_6(-)$	0	0	0	0	0.02	0.3	0.8
$v_7(-)$	0	0	0	0	0.5	0.6	0.03

The result for this initialization (random-test 2):

Cycle →	0	1	2	3	4	5	6
$v_1(+)$	1.00	0.00	0.00	0	0	0	0
$v_2(+)$	1.00	0.00	0.00	0	0	0	0
$v_3(+)$	0	1.00	0.00	0.00	0	0	0
$v_4(*)$	0	0	1.00	0.82	0.25	0	0
$v_5(*)$	0	0	0.01	0.01	1.00	0	0
$v_6(-)$	0	0	0	0	0.10	0.02	1.00
$v_7(-)$	0	0	0	0	0.70	1.00	0.00

Number of iterations : 4
TDDF : 2000, **BUDF :** 2000, **HDF :** 1000
kd : 2, **ku :** 2, **kh :** 6
Library : $\{< multiplier : * >, < adder : + >, < subtractor : - >\}$
Resources : 1 multiplier, 2 adders, 1 subtractor.

Appendix F

User manual

This appendix is concerned with the actual implementation of the fuzzy (pre-)scheduler. The algorithm can be divided in two parts :

- Initialization and Building of the necessary data-structures
- The actual (iterative) algorithm

Before going on with the discussion of the procedures, an outline of the fuzzy data-structure is given.

F.1 Fuzzy data-structure

In figure F.1, an overview of the used data-structure is shown. In figure F.1 (A) a graph to be scheduled is depicted, in figure F.1 (B) the corresponding data-structure is shown. As can be seen in figure F.1 (B), to each node an interval $I(v_i)$ is assigned. The interval $I(v_i) = [InfLow(v_i), InfHigh(v_i)]$ (which is an abbreviation for *Influence*) is the interval defined by :

$$InfLow(v_i) = \begin{cases} 0 & \text{if } ASAP(v_i) - \delta(v_i) + 1 \leq 0 \\ ASAP(v_i) - \delta(v_i) + 1 & \text{if } ASAP(v_i) - \delta(v_i) + 1 > 0 \end{cases} \tag{F.1}$$

$$InfHigh(v_i) = ALAP(v_i) \tag{F.2}$$

The *influence-interval* is used to link per cycle-step the competitor-nodes. For example the nodes v_1 and v_2 are competitors in cycle 0, because they can be mapped onto the same module in the same cycle. Therefore these nodes are linked in this cycle (see igure F.1 (B)). Note, a dashed line between two nodes represents a link, a *NIL*-symbol represents *no* link). As can be seen in figure F.1 (B), each node has *no, one* or *two* neighbour-nodes (i.e. a node where it is linked with).
The borders of the influence interval are chosen with a special reason. Take for instance the situation as depicted in figure F.2 (A), with the assumption that the nodes shown must be executed on the same module-type (it is clear that under these circumstances the nodes shown are *competitors*). The influence intervals for each node are shown in figure F.2 (B). Note that if node v_3 is started in cycle 1, this also affects the nodes v_1 and v_2, although cycle $1 \notin SI(v_1), SI(v_2)$!! In the same way, starting node v_1 or v_2 in cycle 3 will also affect node v_3 (although cycle $3 \notin SI(v_3)$!). Therefore the influence interval is introduced to be able to compute the *horizontal distorsions* ($DIST_{horz}$) the regarded node causes, due to the size of its start-interval and its delay.

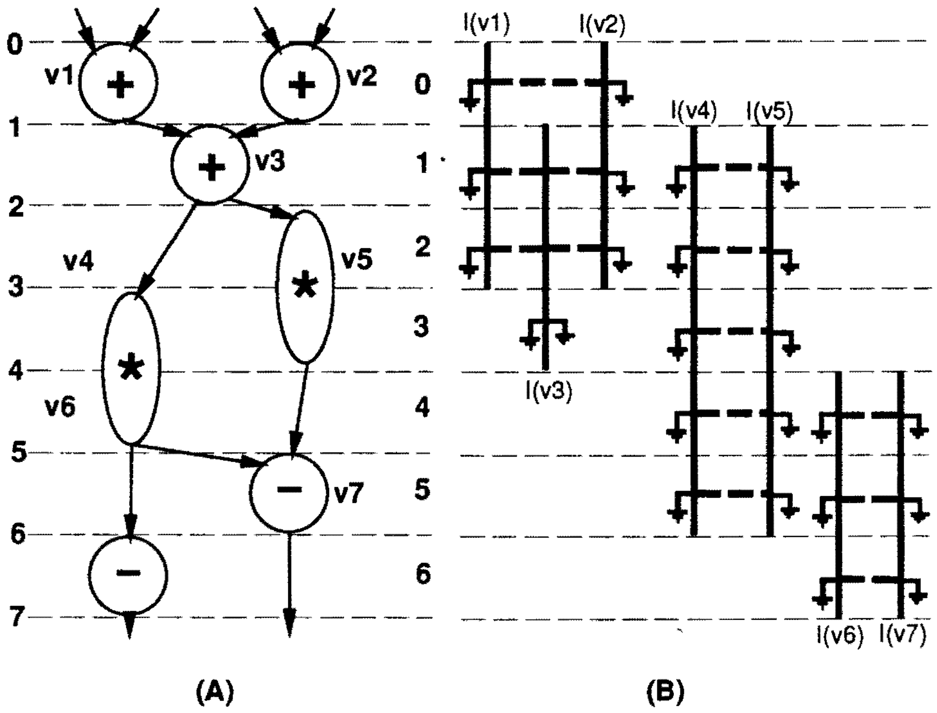

Figure F.1: Used data-structure in implementation.

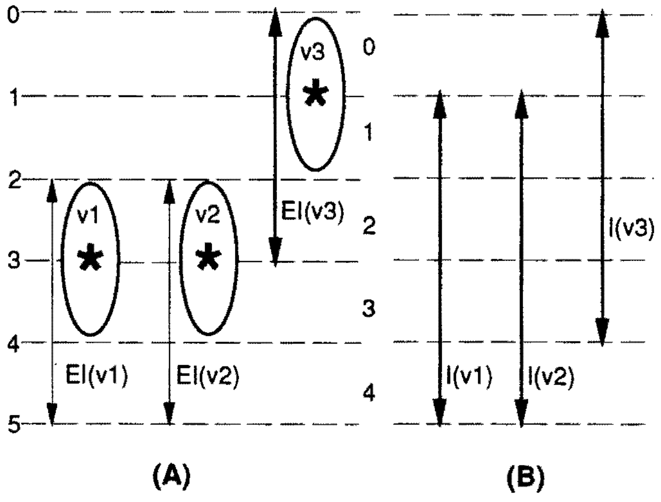

Figure F.2: Example of the use of influence-intervals.

To compute the *vertical distorsions* (i.e. $DIST_{up}$ and $DIST_{down}$) the structure present in the *NEAT*-system are used. That means that for each node in the graph its predecessors and successors can be reached by using the variables *pred-nodes* and *suc-nodes*.

F.2 Description of the functions used

F.2.1 Building and Initialization of data-structures

getBelieve()/setBelieve()
These macro's are used to retrieve or to store the membership-grades. Note that in the implementation *Believe* is written instead of *Membership-grade* !!!

InitStartInterval()
input : a data-flow graph provided with ASAP-ALAP-values
output : a data-flow graph with for each node an influence- and a start-interval created and initialized
description : the function determines the size of the influence-interval and the start-interval. When the sizes of these intervals are known, the function creates an array *infArray* with the size of the influence-interval. Likewise, the start-interval *fuzArray* is created, with the size of the start-interval. After creating the mentioned arrays, the variables in the arrays are initialized.

CreateHorzLinks()
input : a data-flow graph with "unlinked" elements of influence-array
output : a data-flow graph with linked competitor-nodes, for all cycles
description : for each cycle-step the competitor-nodes are linked with each other, such, that the result is a doubly linked list of competitor-nodes for each cycle step.

SatisfiedLinkCond()
input : two data-flow nodes and a cycle-step
output : "1" if the nodes given are competitors, "0" if the nodes given are *no* competitors
description : Given a node *curr* and a cycle-step, check out whether the node *cand* is a competitor of node *curr* in the specified cycle-step.

Det-MaxAccBelDown()
input : a data-flow graph
output : a data-flow graph with computed maximum $DIST_{down}$
description : this function determines the maximum $DIST_{down}$ that can by caused by a node in the data-flow graph regarded. The computed value is used for assigning a value to the damping-parameters in the membership-functions (see section 4.3.1).

Det-MaxAccBelUp()
input : a data-flow graph
output : a data-flow graph with computed maximum $DIST_{up}$
description : this function determines the maximum $DIST_{up}$ that can by caused by a

node in the data-flow graph regarded. The computed value is used for assigning a value to the damping-parameters in the membership-functions (see section 4.3.2).

F.2.2 Actual schedule-functions

FuzzySchedule()
input : a data-flow graph with for each cycle the competitors linked
output : for all nodes, to each cycle of the corresponding start-interval a membership-grade is assigned
description : Given a data-flow graph with the data-structures as created in *CreateHorzLinks()*, the function determines the membership-grades of the cycles (for all nodes). This is done by iteration (see chapter 4), using the functions : *TraverseTopDown()*, *TraverseBottomUp()* and *TraverseHorizontal()*.

TraverseTopDown()
input : a data-flow graph
output : a data-flow graph with computed membership-grades
description : this function determines for each start-position of a data-flow node the $DIST_{down}$ and uses the function $Mu - Down()$ to compute the membership-grades from these distorsions (see also section 4.3.1).

TraverseBottomUp()
input : a data-flow graph
output : a data-flow graph with computed membership-grades
description : this function determines for each start-position of a data-flow node the $DIST_{up}$ and uses the function $Mu - Up()$ to compute the membership-grades from these distorsions (see also section 4.3.2).

TraverseHorizontal()
input : a data-flow graph
output : a data-flow graph with computed membership-grades
description : this function determines for each start-position of a data-flow node the $DIST_{horz}$ and uses the function $Mu - Horz()$ to compute the membership-grades from these distorsions (see also section 4.3.3).

NormAndCompareDown
input : a data-flow node
output : normalized membership-grades for the node
description : while traversing top-down, this function takes the fuzzy intersection of all new computed and old values, and normalizes the result.

NormAndCompareUp
input : a data-flow node
output : normalized membership-grades for the node
description : while traversing bottom-up, this function takes the fuzzy intersection of all new computed and old values, and normalizes the result.

NormCard()
input : a data-flow graph
output : a data-flow graph with normalized membership-grades for all cycles of a data-flow node
description : by the option "-n", the user can define the output to be returned in normalized format. The membership-grades are normalized on the cardinality of the fuzzy set (see equation 3.6)

F.3 Running the program

The program can be run by typing : *gfuz <input-file>*
Typing : *gfuz -h* will show the possible options.
By using the option *+o*, the fuzzy schedule information is written into the nodelinks. In these nodelinks the keywords beginning with *fuz* denote items of the fuzzy schedule results. More specific :

- *fuzbegin* denotes the first cycle in which an operation may be started
- *fuzsize* denotes the number of cycles of the start-interval
- *fuzarray* denotes the membership-grades for each cycle in the start-interval.

F.4 Overview of functions of the fuzzy algorithm

```
FuzDfGraph.* :
==============
  void    InitStartInterval();
  void    CreateHorzLinks();
  void    PrintFuzzyResults();
  double  Det_MaxAccBelDown();
  double  Det_MaxAccBelUp();
  int     SatisfiedLinkCond(FuzDfNode* current, FuzDfNode* cand,double t);
  void    FuzzySchedule(double MaxDown, double MaxUp);
  void    TraverseTopDown(int m, double d_down);
  void    TraverseBottomUp(int m, double d_up);
  void    TraverseHorizontal();


FuzDfNode.* :
=============
  void    setExStart(double i)
  void    setExEnd(double i)
  double  getExStart()
  double  getExEnd()
  void    setInfLow(double i)
  void    setInfHigh(double i)
  double   getInfLow()
  double   getInfHigh()
  int      getMark(int m)
  void     setMark(int m)

  FuzLinksPtr*       fuzArray;
  int      numExecs;    // = size of fuzArray.
  int      numFuzs;     // = num. of start-possibilities.

  InfLinksPtr*       infArray;
  int      influence;   // = size of infArray.
  int      offs;

  double   CardSet;

  void     TraverseTopDown(int m, double d_down);
  void     TraverseBottomUp(int m, double d_up);
  void     Det_MaxAccBelDown(int m);
  void     Det_MaxAccBelUp(int m);
  void     NormAndCompareDown(double d_down);
  void     NormAndCompareUp(double d_up);
  void     NormalizeArray();
  void     NormCard();

  double   Mu_Down(double bel, double d_down);
  double   Mu_Up(double bel, double d_up);
  double   Mu_Horz(double bel,int Imps);


FuzLinks.* :
============
  double getBelieve()
  void   setBelieve(double b)
  int    getConcs()
  void   setConcs(int c)
  double getAccBelDown()
  void   setAccBelDown(double d)
  double getAccBelUp()
  void   setAccBelUp(double d)
  double getStartSucs()
  void   setStartSucs(double s)


InfLinks.* :
============
  InfLinksPtr  getNext()
  InfLinksPtr  getPrev()
```

```
void   setNext(InfLinksPtr n)
void   setPrev(InfLinksPtr p)
int    getIsLinked()
void   setIsLinked()
double getBelieve() // Believe == membership-grade !!
void   setBelieve(double b) // Believe == membership-grade !!
int    getConcs()
void   setConcs(int c)
```