Network and Controller Generation in High Level Synthesis

By Eric K. Lie

Master Thesis

Eindhoven, August 1993
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Dr. Ir. J.T.J. van Eindhoven

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports
Abstract

High Level Synthesis (HLS) automates a part of the chip design trajectory, that has not yet been covered by the traditional silicon compilers. The input of the HLS system is a high level language, describing the behavior of the desired network. The output of a HLS system is a network at register transfer level (RTL) and a description of the controller in that network. This output serves as input for the traditional silicon compilers.

At the Design Automation section, department of Electrical Engineering of the Eindhoven University of Technology, research is conducted on a HLS system called NEAT (New Eindhoven Architectural synthesis Toolbox). NEAT uses a design representation based on the data flow, control and network graphs, and links between nodes of these graphs.

The high level description (input) is immediately transformed into a data flow graph. This graph will be processed by the allocator, scheduler and binder. The allocator allocates network devices for the operations in the data flow graph, the scheduler schedules the operations in the data flow graph and the binder maps operators in the data flow graph on the network modules in the network graph. The result of invoking these three processes is a design with a data flow graph, control graph, network graph and a set of nodelinks. The nodelinks contain the binding information about which operator is implemented by which network module, and the scheduling information about when to execute what operation in the data flow graph. However, the network graph consists of only a set of nodes (network devices) without edges (wiring).

To finish the design, an interconnect and controller generator is needed in the HLS system. The job of the interconnect and controller generator is to finish the network graph by adding edges representing connections between the nodes in the network graph. And also a description of the controller module in the RT-network has to be generated.

The design representation is examined and rules for using the nodelinks in the design to store scheduling and binding information are formulated. We developed a concept, called the delay index, which is very useful to describe these rules. Some of the subprocesses of our interconnect and controller generator are based on the delay index too.
The interconnect and controller generation process has been divided into 5 subprocesses. The interconnect and controller generator has been implemented as a package of programs for each subprocess. The division into subprocesses made it easier to understand and to maintain.
## Contents

1 Introduction ................................. 1
1.1 High Level Synthesis .......................... 2
1.2 Interconnection and Controller Generation ................. 3

2 Data Flow Graph .............................. 5
2.1 Introduction ................................ 5
2.2 Node types ................................ 5
  2.2.1 Input and output nodes .................. 5
  2.2.2 Operation nodes ......................... 6
  2.2.3 Branch and merge nodes ................. 7
  2.2.4 Entry and exit nodes ................... 8
2.3 Edge types ................................ 8
  2.3.1 Datatype and width ..................... 9
2.4 Conditional constructs ........................ 10
2.5 Some definitions ............................ 18
2.6 Validation of Data Flow Graphs ................. 20

3 Control Graph ................................. 22
3.1 Introduction ................................ 22
3.2 Node types ................................ 22
  3.2.1 Input and output nodes .................. 22
  3.2.2 State nodes ............................... 22
  3.2.3 Branch, merge, entry and exit .......... 22
3.3 Control graphs and deterministic finite state machines ........ 23
3.4 Validation of control graphs .................. 28

4 Network Graph ................................. 31
4.1 Introduction ................................ 31
4.2 Combinatoric and sequential devices .................. 31
  4.2.1 Combinatoric devices ................. 31
  4.2.2 Sequential devices ..................... 31
4.3 Node types ................................ 33
  4.3.1 Input, output and inout nodes .......... 33
  4.3.2 Operation nodes ......................... 33
  4.3.3 Interconnection nodes ................... 34
  4.3.4 The controller .......................... 35
## CONTENTS

4.3.5 Delay and timing of network modules ........................................... 36
4.4 Network edge .................................................................................. 38
4.5 Some definitions ............................................................................. 38
4.6 Validation of network graph ........................................................... 39

5 Graphlinks and Nodelinks .................................................................. 40
5.1 Introduction .................................................................................... 40
5.2 Nodelinks ........................................................................................ 40
5.3 Use and interpretation of nodeLinks ............................................... 41
5.3.1 Plain graph ............................................................................... 41
5.3.2 BM-constructs in data flow graph ............................................. 43
5.3.3 BM-constructs in the control graph ......................................... 44
5.3.4 EX-constructs .......................................................................... 49
5.4 Some definitions and restrictions ................................................... 51

6 Modeling the use of nodelinks ............................................................ 54
6.1 Introduction .................................................................................... 54
6.2 Condition analysis on data flow and control graphs ....................... 54
6.2.1 Relations between data flow conditions of data flow edges .......... 56
6.2.2 Calculation of data flow conditions in data flow graph ................. 59
6.2.3 Calculation of data flow conditions in the control graph ............... 63
6.3 Delay index .................................................................................... 67
6.4 Rules for using nodelinks ............................................................... 69

7 Interconnect and controller generator ................................................ 71
7.1 Introduction .................................................................................... 71
7.2 Mapping data flow paths onto network edges .................................. 73
7.2.1 Network conditions ................................................................... 73
7.2.2 Transformation of data flow conditions into network conditions ... 74
7.2.3 Network function ....................................................................... 74
7.2.4 Algorithm .................................................................................. 75
7.3 Solve signal conflicts ..................................................................... 77
7.3.1 Multiplexer .............................................................................. 77
7.3.2 Tri-state .................................................................................... 78
7.4 Interconnection for control purposes .............................................. 81
7.4.1 Extension of network function ............................................... 81
7.4.2 Algorithm ................................................................................ 81
7.5 Extract finite state machine describing the controller ..................... 88
7.5.1 TAB table .............................................................................. 88
7.5.2 Bitvector calculus ..................................................................... 89
7.5.3 Spec’s of control node .............................................................. 91
7.5.4 Expansion of literal function with start condition ....................... 97
7.5.5 Specifying next states .............................................................. 97
7.5.6 Specifying the output bitvectors .............................................. 99
7.5.7 Specifying the ready bitvectors ............................................... 101
7.5.8 Generation of ready signals ..................................................... 104
7.5.9 Disable tri-states ...................................................................... 105
CONTENTS

7.6 Extract Net and Module Lists ........................................... 106

8 Results .............................................................................. 109
  8.1 Data flow and network conditions ..................................... 109
  8.2 Special value nodes in bm constructs ............................... 110
  8.3 Interconnect optimization .............................................. 110
  8.4 Combinatorial delay and chaining .................................... 110

9 Conclusion ........................................................................ 111
Chapter 1

Introduction

Electrical integrated circuits are of enormous importance in the current society. There is a wide range of applications for the chip, ranging from sophisticated microprocessors in computers to application specific chips in for example washing machines. The traject from idea to the production of a chip may cover months of work by a team of specialized engineers. Because the use of chips is getting more common every day, and companies want to decrease the "time to market" of their products, many people have conducted research on automating the design of chips.

Figure 1.1: Scheme of a traditional silicon compiler without High Level Synthesis.

At this moment, there exist programs that generate masks for the production of the chips automatically from a network and controller description. The network is specified in the from of a module and netlist, on register transfer level (RTL-network). A RTL-network consists of modules like registers, ALUs, multipliers, adders, multiplexers, etc. The network usually contains a controller module. This module generates the control signals for the other modules in the network. The controller is specified by a deterministic finite state machine. In figure 1.1 we see a scheme of this traditional silicon compiler. The RTL-network and controller description goes first through the logic synthesis part. Logic synthesis generates a gate level network of the RTL-network including the controller. The gate network is then transformed into the masks, for the production of the chip by layout synthesis.
CHAPTER 1. INTRODUCTION

1.1 High Level Synthesis

Using the traditional silicon compilers, one could shorten the design traject of chips. However, first we have to transform the idea into a RT-netwerk and specify a finite state machine of the controller, before the silicon compiler can be used. At the Design Automation section, department of Electrical Engineering of the Eindhoven University of Technology, research is conducted on automating the traject from idea to RTL-netwerk and controller. This project resulted in a High Level Synthesis (HLS) system, called NEAT [NEAT 92]. Using HLS the silicon compiler can now handle a much higher level of description as input (see figure 1.2). HLS will generate the RT-netwerk and controller from a behavioral or functional description of the wanted chip in some kind of language, like Hardware C, VHDL, Silage, etc.

Figure 1.3 shows us the scheme of the current High Level Synthesis approach in the NEAT-system. The functional description of the chip is first transformed into an internal format, called data flow graph. The data flow graph is a token flow graph with nodes representing operations, and edges representing token transport between the nodes. A design representation in the NEAT system, is based on three graphs:

1. Data Flow Graph
   Represents the behavior of the chip.

2. Control Graph
   Represents the structure of the controller.

3. Network Graph
   Represents the register transfer level network.

The data flow graph is processed by the allocator, which creates the network graph in the design, with nodes representing the allocated hardware modules required to implement the
CHAPTER 1. INTRODUCTION

operations of the data flow graph. In a network graph, the nodes represent network modules and edges represent wiring. The scheduler creates the control graph in the design and specifies when the operators in the data flow graph should be executed. A control graph is also based on the token flow principle. The control graph is used as an intermediate format describing the structure of the finite state machine representing the controller.¹ A design representation contains except the three graphs also a set of nodelinks. The information concerning the scheduling of the data flow nodes is stored in nodelinks. Nodelinks are used to store relationships between the nodes of the three graphs of a design. After scheduling, the binder decides for what purpose (data flow operator) the network modules in the network graph are being used. This binding information is also stored in nodelinks.

1.2 Interconnection and Controller Generation

After passing the binder, the design will be modified by an interconnection and controller generation process, which is the subject of this report. The output of the binder is a design representation, in which we have to finish the network graph by connecting the nodes (modules) to each other with edges (wiring) and extracting the description for the controller in that network. This must be done in such a way that the final RTL-network (represented by

¹To make it easier to understand this, think of the control graph as a finite state machine with the nodes representing the states and edges the transitions between these states. This is not exactly true as we shall see later, but it is good enough for now.
the network graph) with the specified controller are a correct implementation of the behavior described by the data flow graph within the timing constraints given by the control graph.

In chapters 2, 3 and 4 the data flow, control and network graphs is discussed. In chapter 5 is shown how nodelinks are used within a design to specify the relations between the nodes of these three graphs. Because there does not exist any formal models for the use of nodelinks in representing the schedule and binding information, a model is introduced in chapter 6. Using this model requirements are specify for the designs that need to be interconnected. This will result in terms like the delay index, which is very useful in understanding that our construction of the complete network and controller is correct. The interconnection and controller generation processes will be discussed in chapter 7.
Chapter 2

Data Flow Graph

2.1 Introduction

The first step in the High Level Synthesis process is to transform the high level behavior description into a data flow graph. Data flow graphs are based on the token flow principle. In a data flow graph, nodes represent operations and edges represent the transfer of tokens from one port of a node to the port of another node. Each node may contain several input and output ports. The origin (destination) of an edge is always connected to an output (input) port. An input port can only be connected to at most one edge\(^1\). One single data value instance is defined to be a token. Depending on the semantics of a node, the execution of a node requires tokens to arrive through edges on a specific subset of its input ports. The execution of a node is a process where tokens are fetched from the edges on its input ports, and tokens are put on its output ports. When we say that a token is put on an output port, we actually mean that, as many tokens (carrying the same value) are put as there are edges connected to that port, such that each edge gets its own token. The tokens a node puts on the edges connected to its output ports, may then carry the result of the execution of the data flow node.

2.2 Node types

By the difference of semantics and use of data flow nodes, we will distinguish between the following node types.

2.2.1 Input and output nodes

Input and output nodes are used for simple asynchroner communication with the world outside the data flow graph. When synchronization is needed, the operation nodes "get" and "put" should be used. An input node contains one output port and is the only node in a data flow graph that contains no input ports. To execute an input node, it is required that a token is placed on that node. A data flow graph can be executed by placing tokens on all its input nodes. When tokens are placed on the input nodes of a graph, the nodes in the graph are executed, until no nodes can be executed anymore.

\(^1\)This is only true if we do not consider timing edges, see section 2.3
CHAPTER 2. DATA FLOW GRAPH

Output nodes contain one input port and are the only nodes in a data flow graph containing no output ports. So, tokens arriving at an output node, will not lead to the creation of other tokens.

2.2.2 Operation nodes

An operation node will execute when

- all edges on its input ports contain a token.
- all edges on its output ports contain no tokens.

Operations can be arithmetic, like *, +, ++, or boolean like V, <, =, or any other complex function. Operation nodes have at least one input port and at least one output port. When a data flow node of type "operation" executes, it will perform the operations it represents on the values of the incoming tokens. The results will be carried by tokens, that are put on the output ports. Notice that the values of the tokens on different output ports do not have to be the same. There are many types of operation nodes. We will discuss here a few predefined operation nodes.

- **Unary arithmetic operations** like ++ and -- have one input port and one output port. The operation node ++ (--) will put tokens on its output node carrying the value equals to the numerical value of the incoming token plus (minus) one.

- **Binary arithmetic operations** like x, +, -, / and many others, are represented by data flow nodes with two input ports for the operands and one output port for the result. These operations are performed on the numerical values carried by the incoming tokens.

- **Binary boolean operation** like V, <, =, and others have the same ports as the binary numerical operation. The result carried by the outgoing tokens of boolean operations is either false or true (0 or 1).

- **Constant nodes**, have one input port and one output port. All the outgoing tokens from one constant node, have the same value. This value is a constant and specified in the constant node. Thus, the incoming token is only required to execute the constant node. The value carried by the incoming token is of no importance for the constant node.

- **Value nodes**, have one input and one output port. The outgoing tokens all carry the same value, which is the same as the value carried by the last incoming token. So, in fact no operation is performed on the values of the incoming tokens. Value nodes have a special meaning in data flow graphs, they are introduced in the data flow graph by the scheduler to indicate that in the final network possibly, a value has to be stored for later use. We will discuss this in later sections.

- **Get nodes**, have one data output port and a input port and output port for synchronization purposes. A get node can be used for communication from the data flow graph to the world through the data port on handshake basis.
CHAPTER 2. DATA FLOW GRAPH

- **Put** nodes, have one data input port and a input port and output port for synchronization purposes. A put node may be used for communication from the outside world into the data flow graph through the data port on handshake basis.

Each operation node in the data flow graph must be instantiated from a data flow graph in the NEAT database. A standard library has been created containing the data flow graphs for the predefined node-types mentioned above. For example when we create a node of type "\(*\)" in a data flow graph, the system will search in the database for a data flow graph named "\(*\)", and we will get a node with as many input and output ports as there are input and output nodes in the data flow graph "\(*\)". The input and output ports in the created node will be named after the input and output nodes in the graph "\(*\)". In this way all data flow nodes of type "\(*\)" will have the same port configuration. Using this hierarchical feature of the NEAT-system, we may also create our own operation node, say \(T\), by first creating a data flow graph named \(T\).

### 2.2.3 Branch and merge nodes

A branch node has two input ports (two incoming edges): a "control" port and a "data" port, and it has at least two "data" output ports. These output ports are referred to by 0, 1, \ldots, \(n\), with \(n\) the number of output ports. A selection list is attached to each branch node. This list specifies on which output port tokens should be put given the value of the token on the control port. When the selection list is not specified, a default selection list is assumed. In which case, the numerical value of the token on the control port corresponds to the name of the data input port. A branch node executes whenever tokens are available on the data input port and control port, and there are no tokens on the outgoing edges. Execution of a branch node implies the evaluation of the selection list with the bitvector representation of the value, carried by the token on the control port. The return value of the selection function then indicates on which output port the token on the data port should be generated. All edges connected to that output port will get exactly one token carrying the same value as the token on the data input port. No tokens are put on the other output ports.

A merge node has one "data" output port, one "control" input port, and at least two "data" input ports. The data input ports are named 0, 1, \ldots, \(n\), while \(n + 1\) is the number of input ports in the merge node. Just like the branch node, a selection list is attached to each merge node to specify from which data input port, a token should be taken. The default selection list is applied when no selection list is specified. The numerical value of the token on the "control" port then indicates the name of the data input port. The execution of a merge node requires:

- one token on the edge of the control port.
- one token on the edge of the data input port indicated by the selection list
- no tokens on all outgoing edges.

When these requirements are met, the merge node executes and puts on all outgoing edges tokens with the same value as that of the token on the data input port, indicated by the selection function.
2.2.4 Entry and exit nodes

Using the nodes presented previously, we could build cyclic data flow graphs. However, we do not allow that. The only legal way to bring loops into data flow graphs is by using entry exit constructions. An entry node has the same semantics as the merge node. Thus, it has one control port, one data output port, at least two data input ports 0, 1, ... and of course a selection list. The same resemblance also exists between an exit node and a branch node.

2.3 Edge types

In the original data flow graph semantics [Ejndhoven 91], many types of data flow edges are presented. For our purposes, some of these edges are not relevant, while distinction between the relevant edges are not really necessary.

Timing edge These edges are originally used for the indication of timing constraints and to enforce a required execution ordering between data flow nodes. Since we are only interested in scheduled data flow graphs, the timing and execution order is already determined by the combination of data flow and control graphs. Thus, timing edges are not required for our purposes and we will assume that deletion of these edges from our data flow graphs will not affect the behavior described by the data flow and control graph. This is true because the data flow graph that we receive has been scheduled, so that the timing information is stored in the nodelinks already.

Data edges Data edges are used to transport tokens carrying values that have to be processed by the nodes in the graph. These edges are the default type of data flow edges, and this is the most important kind of edges for us. If we just speak about edges in a data flow graph, we will mean data edges.

Control edges The data flow edges, connected to the control ports of branch, merge, entry and exit nodes, are of type "control". In fact these edges may be regarded as data edges because the tokens transported by these edges are also carrying values that have to be processed by data flow nodes. The idea for the distinction between control and data edges is to make it easier to emphasize the control edges in drawings. But for our purposes, this distinction is not required, thus we will refer to these edges in the same way as to data edges.

Source edges Edges connected to the input ports of "constant" nodes, must be source edges. These edges were introduced, because the tokens through these edges are only used to activate the "constant" nodes, while the value carried by the tokens are not processed. Since the data flow graphs, that we receive as input for our interconnect and controller generator, are scheduled, the information about when to activate the constant node is stored in some other way already. However, the source edge is still useful in determining the condition (section 6.2) under which the constant node is executed.
CHAPTER 2. DATA FLOW GRAPH

Chain edges Chain edges are used to connect the input and output synchronizing ports of the get and put nodes in the data flow graph. These edges are required in the original data flow graph to indicate that the get and put nodes that are connected to each other by chain edges should use the same io-device in the network. Another purpose of these chain edges is to indicate a precedence relation between the execution of the put and get nodes. However, we will use module bound and scheduled data flow graphs. In these graphs the information about what device is used by what data flow operation and the information about precedence relations between the operations have already been stored in some other way (nodelinks). Thus, we could delete the chain edges from the data flow graphs, without affecting the design. But we will keep the chain edges in the data flow graphs, because using these edges, it is easier to make explicit, the conditions for the execution of the get and put nodes (section 6.2). Since execution of an operation node requires that the incoming edges of these nodes carry tokens, it's easier to find the execution condition if the chain edges are kept in the data flow graph. So, the chain edges are required by us for the same purpose as the source edges. We will therefore make no distinction between the source and chain edges, and refer to chain edges as "source" edges.

2.3.1 Datatype and width

The value of the tokens transported by "data" edges (including the "control" edges) usually are relevant, since operations will be performed on these values. The datatype and width are properties of data flow edges, that makes it possible to interpret the value of the token in several domains (numeric, bitvector, boolean).

- **unsigned integer**
  The numerical value of an unsigned integer of width \( w \) is given by:

  \[
  value = \sum_{i=0}^{w-1} bit[i] \cdot 2^i
  \]

- **signed magnitude integer**
  The numerical value of a signed magnitude integer of width \( w \) is given by:

  \[
  value = sign \cdot \sum_{i=0}^{w-2} bit[i] \cdot 2^i
  \]
  with \( sign = 1 \) if \( bit[w - 1] = 0 \), and \( sign = -1 \) if \( bit[w - 1] = 1 \).

- **two complement**
  The numerical value of a two complement integer of width \( w \) is given by:

  \[
  value = \sum_{i=0}^{w-2} bit[i] \cdot 2^i - bit[w - 1] \cdot 2^{w-1}
  \]
• **boolean**

The boolean value of a boolean of width \( w \) is given by:

\[
value = \begin{cases} 
  \text{true} & \text{if } \bigwedge_{i=0}^{w-1} (\text{bit}[i] = 1) \\
  \text{false} & \text{if } \bigwedge_{i=0}^{w-1} (\text{bit}[i] = 0)
\end{cases}
\]

For all other cases the boolean value is unspecified.

As we see, the bitvector representation is the most general representation of a value. It is possible to transform all the representations into bitvectors. Because a bitvector may have unspecified bits (don't-cares), it is not always possible to transform a bitvector into one value using the given representations. The unsigned integer representation is the default datatype for a data flow edge.

When the value carried by some token has a width that does not match with the width of the port where the token should go through, a transformation is carried out on that value. The transformation is always performed in the bitvector domain and depends on the original datatype. Here is the table for the transformation of bitvectors into bitvectors of another width,

<table>
<thead>
<tr>
<th>Datatype</th>
<th>( p &lt; w )</th>
<th>( p &gt; w )</th>
</tr>
</thead>
</table>
| unsigned integer or      | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < p) 
| boolean                   |                                                  | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < w) 
|                           | \( \text{port}[i] = 0 \ (w \leq i < p) \)        |                                                  |
| two-complement integer    | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < p) 
|                           |                                                  | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < w) 
|                           |                                                  | \( \text{port}[i] = \text{bit}[w - 1] \ (w \leq i < p) \) |
| signed magnitude integer  | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < p - 1) 
|                           | \( \text{port}[:p-1] = \text{bit}[w - 1] \)      | \( \text{port}[i] = \text{bit}[i] \ (0 \leq i < w - 1) 
|                           |                                                | \( \text{port}[i] = 0 \ (w - 1 \leq i < p - 1) 
|                           |                                                | \( \text{port}[p - 1] = \text{bit}[w - 1] \)      |

In this table a bitvector, specified by \( w \) bits (\( \text{bit}[i] \)) is transformed into a bitvector of \( p \) bits (\( \text{port}[i] \)).

### 2.4 Conditional constructs

Nodes of the types "branch", "merge", "entry" and "exit" (bmex), are called conditional nodes. Using these conditional nodes it is possible to build conditional constructs.

There are two kinds of conditional constructs: branch merge (bm) and entry exit (ex). Branch and merge nodes are used to build case and if then else constructions like that in figure 2.1. Entry exit constructs are used to build loops. An example of the use of entry exit constructions
process if4(a, b, c, z)
in int a, b, c;
out int z;
{
    if (c > 0)
        z = a - b;
    else
        z = a + b;
}

Figure 2.1: a. Hardware C description with if then else construction. b. Data flow graph constructed from the hardware C description.
to implement a while loop in a data flow graph, is depicted in figure 2.2. We see that there is a causality problem with the token for the control ports of the entry nodes, since this token is generated after the entry node has been passed. This problem is solved by initially putting a token on the edges to the control ports of the entry nodes, carrying a value corresponding to the $0^{th}$ port of the entry nodes, according to their selection lists.

![Data flow graph](image)

Figure 2.2: a. Hardware C description with while construction. b. Data flow graph constructed from the hardware C description.

Notice that there may be several nodes that generate the token for the condition nodes of one construct. For example the last condition construct of the data flow graph presented in figure 2.3 gets its control tokens from the constant node or from the input port. These "conditional conditions" are allowed, as long as the edges to the control ports of the conditional nodes of one conditional construct, have the same origin port. There is only one case allowed, in which the edges to the control ports of the condition nodes of one construct is connected to different origin ports. For example, figure 2.4 shows us the data flow graph of figure 2.1 with a small change. The merge node of the construct receives now its control token from the value node, while the branch nodes still get their control token from the $>$-operator. We will call such a conditional construction a construction with a special value node. Thus, it is even possible that the condition nodes of one construct get their control tokens directly from different nodes. In the original data flow graph semantics, it is not allowed that the control ports of the condition nodes of one conditional constructs are connected to edges with different origin ports. We will show later (section 5.3.2) that this could be a too severe restriction, by which some good schedules cannot be represented easily in the NEAT-system.
In order to build an unrestricted interconnector as possible, we will allow the special value nodes in constructions like that in figure 2.4.

However, there are some basic restrictions on the use of conditional constructs that we would like to maintain.

- The selection list of a conditional node may not be ambiguous. This means that there must not exist a value for the control token, such that it corresponds to more than one data port according to the selection list.

- The selection lists of the condition nodes of one condition construct must be the same.

- The bodies of bm constructs must be closed, i.e. there are no edges between the bodies. In figure 2.5 the bodies of a certain bm construct in a data flow graph is given. We see that there are in this case three conditional bodies 0, 1 and 2. Furthermore there is a test body, which is the part of the data flow graph outside the bm construct. As shown in the picture there must be no edges between the bodies. Notice that the special value node connected to the merge nodes is optional, and may be replaced by a direct connection.

- The bodies of ex constructs must also be closed. Just like the bm constructs, we can also define the bodies of an ex construct. Figure 2.6 shows such an example. Again it is not allowed to have edges between the bodies. The value node connected to the entry nodes is optional, and may be replaced by a direct connection.
Figure 2.4: Data flow graph with different origin ports for the control edges to one construct.
Figure 2.5: The bodies induced by a bm construct in a data flow graph.
Figure 2.6: The bodies induced by an ex construct in a data flow graph.
CHAPTER 2. DATA FLOW GRAPH

Notice that these restrictions do not form a severe restriction for the possible schedules and bindings that can be represented with data flow graphs. On the contrary these restrictions give us more orderly and structured data flow graphs.
2.5 Some definitions

In this section we will give some definitions related to the data flow graph. In the object oriented NEAT environment, data flow graphs contain a set of data flow nodes and data flow edges. Data flow nodes contain a set of input and output ports and data flow edges are a 2-tuple consisting of one origin data flow port and one destination data flow port. Figure 2.7 shows the owner relation between these objects.

![Diagram](image)

Figure 2.7: The owner relation in the object Graph

The definitions we give here, are not only applicable to the data flow objects. Since the control and network graphs are also graphs and constructed with the same basic objects (ports, nodes and edges) as the data flow graphs, we will leave the words "data flow" out of the definitions.

**Definition 2.1 (Nodes in a graph)**

Let $g$ be a graph, then $g.Nodes$ denotes the set of nodes in the graph $g$.

**Definition 2.2 (Nodes in a graph)**

Let $g$ be a graph, then $g.Edges$ denotes the set of edges in the graph $g$.

**Definition 2.3 (Type of a node)**

Let $n$ be a node, then $n.type$ is the type of the node $n$.

**Definition 2.4 (Instantation)**

Let $n$ be a node and $g$ be a graph from which $n$ is instantiated. Then $g$ may be referred to as $n.instedFrom$.

**Definition 2.5 (Ports of a node)**

Let $n$ be a node, then $n.Ports$ is the set of ports contained in $n$. And $n.inPorts$ denotes the set of all input ports contained in $n$. And $n.outPorts$ denotes the set of all output ports contained in $n$.
Definition 2.6 (Port of a node)
Let \( n \) be a node and \( i \) be the name of a port \( p \) in that node. Then port \( p \) may be referred to as \( n.port(i) \).

Definition 2.7 (Owner of a port)
Let \( port \) be a port, then the node in which \( port \) is located is denoted by \( port.owner \).

Definition 2.8 (Owner of an edge)
Let \( g \) be a graph and \( e \) be an edge, such that \( e \in g.Edges \). Then graph \( g \) may be referred to as \( e.owner \).

Definition 2.9 (Edges connected to a port)
Let \( port \) be a port, then the operator \( port.Edges \) is defined as the set of edges connected to \( port \).

Definition 2.10 (Edge connected to a port)
Let \( port \) be a port and \( \#port.Edges = 1 \) and \( e \in port.Edges \). Then \( e \) may also be denoted as \( port.edge \).

Definition 2.11 (Origin and destination ports of an edge)
Let \( e \) be an edge, then \( e.origport \) is the origin port of \( e \) and \( e.destport \) is the destination port of \( e \).

Definition 2.12 (Origin and destination nodes of an edge)
Let \( e \) be an edge, then the operators \( .orig \) and \( .dest \) are defined for the edge \( e \) as
\[
\begin{align*}
  e.orig &= e.origport.owner \\
  e.dest &= e.destport.owner
\end{align*}
\]

Definition 2.13 (Pins of port)
A port actually consists of a set of pins. Let \( port \) be a port. Then its set of pins is denoted as \( port.Pins \).

Definition 2.14 (Pin of port)
Let \( port \) be a port, then a pin of \( port \) may be referred to as \( port.i \) (\( 0 \leq i < \#port.Pins \)). The pin \( port.0 \) corresponds to the least significant bit.

Definition 2.15 (Predecessors of a node)
The function \( pred \) returns a set of all the predecessors of a node. Let \( n \) be a node, then \( pred \) is defined on \( n \) as
\[
pred(n) = (\cup : port \in n.inPorts \land e \in port.Edges : \{e.orig\})
\]
CHAPTER 2. DATA FLOW GRAPH

2.6 Validation of Data Flow Graphs

Using the definitions presented in the previous sections, we can now give the algorithms for validating a data flow graph. By calling the function check algorithm 2.1 on a data flow graph, we can verify whether the data flow graph is valid. DfGraph::check first calls the function acyclic, which return -1 if the data flow graph is not acyclic, when we remove all edges to the input ports of the "entry" nodes in the DfGraph. acyclic returns 0 otherwise. The function closedBodies verifies whether the restrictions on the use of conditional constructs (see page 13) are met. 0 is returned if this is the case, otherwise -1 is returned by closedBodies. The check on the nodes, first calls the check on the ports, and then verifies that at least one data input (output) port of the branch and exit (merge and entry) nodes are used. The check on the ports, makes sure that all input ports, except some, are connected. The check on the edges, verifies that the origin ports are output ports and the destination ports are input ports. (The prefix # for a set of something, denotes the number of elements in that set.)

Algorithm 2.1 (Check the data flow graph)

```c
int DfGraph::check()
{
    /* Pre: this = dfg */
    /* Post: returns -1 if dfg is not valid, 0 otherwise */
    if (dfg.acyclic()) return -1
    if (dfg.closedBodies()) return -1
    for all (dfn ∈ dfg.Nodes)
    {
        if (dfn.check) return -1
    }
    end all
    for all (dfe ∈ dfg.Edges)
    {
        if (dfe.check) return -1
    }
    end all
    return 0
}
```
Algorithm 2.2 (Check the data flow node)

```cpp
int DfNode::check()
{
    /* Pre: this = dfn */
    for all (port ∈ dfn.Nodes)
    {
        if (port.check) return -1
    }

    if (dfn.type = "branch"|"exit")
    {
        if (∃ : port ∈ dfn.inPorts, port.name = "control", 1 = #port.edges) return -1
        if (∃ : port ∈ dfn.outPorts : 1 ≤ #port.edges) return -1
    }

    return 0
}
```

Algorithm 2.3 (Check the data flow port)

```cpp
int DfPort::check()
{
    /* Pre: this = dfp */

    if (((dfp.type = "input") ∧ (dfp.name = "control") ∧ (#dfp.edges ≠ 1)) return -1
    if (((dfp.type = "input") ∧ (dfp.owner.type = "merge"|"entry") ∧ (#dfp.edges > 1)) return -1
    if (((dfp.type = "input") ∧ (dfp.owner.type ≠ "merge"|"entry") ∧ (#dfp.edges ≠ 1)) return -1

    return 0
}
```

Algorithm 2.4 (Check the data flow edge)

```cpp
int DfEdge::check()
{
    /* Pre: this = dfe */

    if (dfe.origport.type ≠ "output") return -1
    if (dfe.destport.type ≠ "input") return -1

    return 0
}
```
Chapter 3

Control Graph

3.1 Introduction

One of the processes in HLS is to schedule the operations represented by the data flow nodes. The control graph is constructed by the scheduler to express the timing of the execution of the nodes in a data flow graph. The control graph will give us the structure of the finite state machine describing the controller, in the final RTL-network. Control graphs are like data flow graphs, based on the token flow principle, and consist of directed edges and nodes, containing several input and output ports. But it has less types of nodes, no different types of edges, and the values carried by tokens have no meaning at all.

We will first discuss the graph aspects of the control graph, and then we will consider how a control graph can be transformed into a finite state machine describing the controller.

3.2 Node types

3.2.1 Input and output nodes

The input and output nodes, have the same semantics and ports, as the input and output nodes presented in section 2.2. A control graph may be executed by putting tokens on its input nodes.

3.2.2 State nodes

A state node contains one input port and one output port. When a token reaches the input port of a state node, then that node will execute, by fetching the token from its input port, and putting a token on the output port after execution.

3.2.3 Branch, merge, entry and exit

Branch, merge, entry and exit nodes may be used to build conditional constructs in the control graph. The restriction on the use of these constructs as presented for the data flow graphs in 2.4, are also required for control graphs. Except that the control ports of conditional nodes in a control graph are not connected to any edges. Conditional nodes in the control graph are virtually connected to conditional nodes in the data flow graph. A conditional node in the
CHAPTER 3. CONTROL GRAPH

control graph will receive a control token at its control port, whenever the virtually connected condition node in the data flow graph receives one. Of course, these tokens will carry the same value and the selection function of both condition nodes must be the same. The link between condition nodes in the data flow graph and those in the control graph will be discussed more elaborate in later sections.

3.3 Control graphs and deterministic finite state machines

The control graph may be seen as an intermediate structure in generating the final controller specifications. The controller module in the final network has to control the modules in the network such that the final network including the controller is a correct implementation of the data flow graph. To keep the transformation of the control graph into a deterministic finite state machine (dfsm) simple, we require that all output ports of the nodes in a control graph are connected to exactly one edge. Control graphs with this restriction have exactly one input and one output node. These graphs have the property that from the moment the graph is executed till the end of execution at most one state node is executing. Under the assumption of course that no extra tokens are put into the control graph through the input node, during execution of the graph.

In figure 3.1 a control graph is transformed into a dfsm. We see that each state node in the control graph is a state in the dfsm. The input and output nodes are mapped on the standby state in the machine. When a signal start is received by the dfsm, the next state is the state corresponding to the first state node in the control graph. So, giving a start signal to the dfsm is the same as placing a token on the input node of the control graph. After a start,
the dfsm will execute and get into the standby state and stays there, until a start signal is received. In figure 3.2 a timing diagram is presented for the controller of figure 3.1b. As we see, state transitions only happen on the rising clock flank. The signals determining the next state are evaluated on the rising clock flank. In our case the signal start is the only relevant signal in determining the next state.

Note that whenever the dfsm receives a start signal the dfsm will start over again with the execution of the control graph (for instance at cycles $T_3$, $T_4$ and $T_5$). At this point the dfsm is not exactly behaving like the control graph. This difference is caused by the fact that only one state in the dfsm is active at a time. If we wish to have more than one state node executing at a moment, we can still transform the control graph into a dfsm. But the states in the dfsm then correspond to combinations of state nodes instead of one state node. In this way the dfsm could grow exponentially. Another alternative is to use another kind of representation which is specially designed for the specification of multiple state controllers. The control graph for instance is such a representation for multiple state controllers. However, at this moment no tools are available for the synthesis of multiple state controllers. So, we will only concentrate on controllers based on dfsm's.

Except an input pin for the start signal, the controller will also need an output pin for a ready signal. This signal is used to indicate that the dfsm finishes (or has finished) the execution of the control graph. Thus, the ready signal is generated in the standby state and the state corresponding to the last state nodes (S3 in figure 3.1) in the control graph. This is also depicted in the diagram of figure 3.2.

When condition nodes (branch, merge, entry, exit) are present in the control graph, the controller will have more input signals. These input signals are needed for the dfsm to determine the next state for each state.

In figure 3.3 a control graph with a branch merge construction is transformed into a dfsm. The control ports of the branch and merge nodes are virtually connected to some control ports in the data flow graph. (How, and what this exactly means will be explained later.) In figure 3.3 the value of the token that virtually arrives at the control ports in the control graph is referred to with the variable select. Again the input and output nodes of the control
Figure 3.3: Transformation of a control graph with branch and merge node into a deterministic finite state machine: a. control graph b. finite state machine

deterministic finite state machine

When the signal start is received, the dfsm must also look at the signal select to determine the next state. When no start is generated (start), the signal select is not important, because there is only one possible next state. Notice that in this specific case the signal ready is generated, during all states of the dfsm.

In figures 3.4, 3.6 and 3.7 control graphs with entry and exit nodes are transformed into dfsm's. In figure 3.5 a timing diagram is presented for the controller of figure 3.4. We see that the signals start and select are both needed to determine the next states. Another important aspect is that the ready signal is not always generated during the last state (S2). This is because whether S2 is the last state depends on the signal select. Obviously to generate a ready signal during some state the following should be satisfied in that state:

1. if start \text{ and the next state is determined to be the standby state.}
2. if start \text{ and the next state would have been the standby state if } \overline{\text{start}}.

One can see in figure 3.5 that for instance in the cycles T_2, T_{11}, and T_{12} the ready signal is generated in state S_2 only if

\[(\text{start} \land \text{select} \lor \text{start} \land \overline{\text{select}}) \equiv \text{select}\]

Notice that the test body of the control graph in figure 3.7 contains no state nodes. The result of this, is that start \land select will always be false, because the entry node gets initially a token such that select is true, thus the dfsm will always stay in the standby state. To avoid this problem, we advise to put at least one state node in the testbody of each entry exit construction.
CHAPTER 3. CONTROL GRAPH

Figure 3.4: Transformation of a control graph with entry and exit node into a deterministic finite state machine: a. control graph b. finite state machine

Figure 3.5: Timing diagram for the controller (dfsm) of figure 3.4b.
CHAPTER 3. CONTROL GRAPH

Figure 3.6: Transformation of a control graph with entry and exit construction with empty body 1, into a deterministic finite state machine: a. control graph b. finite state machine

Figure 3.7: Transformation of a control graph with entry and exit construction with empty test body, into a deterministic finite state machine: a. control graph b. finite state machine
When there are conditional nodes in the control graph, then these nodes could contain not completely specified selection lists. According to the token formalism, when a token arrives at the control port of a conditional node, carrying a value not specified in the selection list, that conditional node should not execute. However, this could be a problem when the control graph is transformed into a finite state machine. Because the result of a finite state machine receiving inputs that have not been specified is unknown. This problem could also happen when for example there is a branch node with an unconnected output. To overcome this problem an error state is introduced in the finite state machine. In this way we can make all unspecified transitions, transit to the error state. In figure 3.8 we see an example of a control graph in which not all transitions are defined. The undefined transition now leads to the error state in the dfsm. Once the error state has been entered, the dfsm will stay there until a new start signal is given to it.

![Control Graph](image)

**Figure 3.8: Transformation of control graph into dfsm with error state**

### 3.4 Validation of control graphs

Using the nodes presented in the section 3.2, it is possible to construct all kinds of control flow graphs. But not all of them will be regarded as valid control flow graphs, suitable for the transformation into a dfsm. The algorithms 3.1, 3.2, 3.3 and 3.4 verifies whether a control graph met the requirements to be a valid control flow graph, such that it can be processed by our interconnector. In the following algorithms, we have applied the definitions on page 18 to the control graph.

Notice that the 
::check functions are the same as those for the data flow objects, except CtPort::check. In CtPort::check we now verify that no edge is connected to the control ports of the conditional nodes and that usually exactly one edge is connected to the output ports.
Algorithm 3.1 (Check the control graph)

```cpp
int CtGraph::check()
{
  /* Pre: this = ctg */
  if (ctg.acyclic()) return -1
  if (ctg.closedBodies()) return -1
  for all (ctn ∈ ctg.Nodes)
  {
    if (ctn.check) return -1
  }
  end all
  for all (cte ∈ ctg.Edges)
  {
    if (cte.check) return -1
  }
  end all
  return 0
}
```

Algorithm 3.2 (Check the control node)

```cpp
int CtNode::check()
{
  /* Pre: this = ctn */
  for all (port ∈ ctn.Nodes)
  {
    if (port.check) return -1
  }
  end all
  if (ctn.type = "branch"|"exit")
  {
    if ¬(∃: port ∈ ctn.inPorts\n    port.name ≠ "control": 1 = #port.edges) return -1
    if (ctn.type = "merge"|"entry")
    {
      if ¬(∃: port ∈ ctn.outPorts: 1 ≤ #port.edges) return -1
    }
  }
  return 0
}
```
Algorithm 3.3 (Check the data flow port)

```cpp
int CtPort::check()
{
    /* Pre: this = ctp */
    if ((ctp.type = "branch"|"merge"|"entry"|"exit")
        (ctp.name = "control") ∧ (#ctp.edges ≠ 0)) return -1
    if ((ctp.type = "input") ∧
        (ctp.owner.type = "merge"|"entry") ∧ (#ctp.edges > 1)) return -1
    if (ctp.owner.type ≠ "merge"|"entry") ∧ (#ctp.edges ≠ 1) return -1
    return 0
}
```

Algorithm 3.4 (Check the control edge)

```cpp
int CtEdge::check()
{
    /* Pre: this = cte */
    if (cte.origport.type ≠ "output") return -1
    if (cte.destport.type ≠ "input") return -1
    return 0
}
```
Chapter 4

Network Graph

4.1 Introduction

A network graph is used as a representation of a circuit at Register Transfer Level. In a network graph, the nodes represent network modules, and edges between them represent signal transport between the modules in the direction of the edge. Each node consists of input, output and inout-ports.

4.2 Combinatoric and sequential devices

All nodes (except the node representing the controller) in a network graph represent some device which is either combinatoric or sequential. The main difference between combinatoric and sequential devices is the presence of buffers (registers) in sequential devices, while combinatoric devices only contain combinatoric logic. We will discuss in this section how these two kinds of devices behave and what is required to assure a proper functioning of these devices.

4.2.1 Combinatoric devices

A combinatoric module may be operated at any moment as required. As soon as the input signals are stable, the outputs will produce the expected signals automatically, after some delay. To maintain the output signals it is required that the input signals do not alter. It is assumed that the delay of combinatoric modules are always less than the cycle-time of the system clock in the network. It is also required that the execution never exceeds a cycle transition (positive flank of the system clock).

4.2.2 Sequential devices

All sequential devices in a network are synchronized by a system clock. Thus, sequential modules always need the system clock for operation. Except the signal clock, sequential modules also have input ports for a signal start. At each positive flank of the system clock, all input signals (except clock and start of course) will be clocked into the input-buffers of the module, if start = 1 at the positive clock flank. Because all input data signals are stored inside the device, the output signals produced after some delay, will be maintained as long as the start signal stays low at every positive clock-flank. The delay of sequential devices may be more
Some sequential modules may have an output port for a ready signal. In that case the module will make the ready signal high (1), to indicate that the performed operation has finished and that before the first next rising clock flank the output signals are valid. The ready signal is especially required when the delay of the operation performed by a module is data-dependent, such that the time needed for the operation is not known in advance. Another application of the ready signal is to connect the ready port of one module to the start port of another module. In this way, when the first module finishes, it will automatically start up the next module without intervention of the controller. This could lead to simpler controllers and less wiring. However, we will not make use of this feature, all control signals such as ready and start will be connected directly to the network's controller.

![Timing diagram for a sequential device with unknown delay.](image)

In figure 4.1 the timing diagram of a sequential device with an unknown delay is given. Whenever the ready signal is generated by the device at some rising clock flank, the data output of the device may be clocked into some other sequential device at the same clock flank. Typically this is the case at the end of cycle T7. We see that the device has generated the signal ready in cycle T7, this means that the output signals should be valid before the end of cycle T7. We see that just before the end of T7 the output signals become valid indeed. Thus, at the transition from cycle T7 to T8 other sequential devices may clock those output signals into their internal buffers. This could be very wise, because a start signal is given to the device at the transition from cycle T7 to T8, so that the output signals are not valid anymore after the transition. Notice that at the end of T3 and T4 the device will clock in the signals on its ports and execution is performed again from the beginning, while the device has not completed its previous execution. In cycles T11 and T12 we see that the output signals will be maintained as long as no start is given to the device.
CHAPTER 4. NETWORK GRAPH

4.3 Node types

4.3.1 Input, output and inout nodes

The input, output and inout nodes are used for the communication with the world outside the network graph. The input node contains one output port. The output node contains one input port. And the inout node contains one inout port. The inout port may be connected to the origin as well as the destination of network edges. The input, output and inout modules are combinatoric modules. These nodes represent uncontrolled terminals. If more synchronized communication is required, then the operation modules, input-pads, output-pads and io-pads should be used.

4.3.2 Operation nodes

The operation nodes represent modules that can perform operations given by the operation data flow nodes. Each operation node in the network graph must be instantiated from a network graph in the NEAT database. A standard library has been created containing the network graphs that define the operation types:

- **Multiplier**
  A multiplier may perform multiplication between the values on the data input ports, and the result is put on the output data port. Except multiplication, the multiplier can also carry out division between the two operands. To control the mode (multiplication or division) of the multiplier, an input port is available. By setting the proper signal on this port the multiplier will operate in the mode we require. The multiplier is a sequential module. Thus, ports for the signals $\text{clock}$ and $\text{start}$ are available. Also there is an output port for the signal $\text{ready}$. Since the delay for operating a multiplier is known in advance, this signal does not serve any purpose for our applications.

- **Adder**
  The adder may perform addition or subtraction between the values on the data input ports. The operation may be chosen by setting the right signal on a control port of the adder. The adder is a combinatoric module, so its delay is less than the system clock's cycletime.

- **Arithmetic Logic Unit**
  The ALU is a sequential module and may be used for many operations, like multiplication and addition between the values on the data input ports. The required operation may be chosen by setting the appropriate signals on a control port of the ALU. The ALU is a sequential device with ports for the signals $\text{clock}$, $\text{start}$ and $\text{ready}$. Just like the multiplier the $\text{ready}$ signal is not really required because the delay of each operation performed by an ALU is known in advance.

- **Constant**
  The constant node contains only one output port. On this port a constant value is set continuously. What constant value is produced, is a property attached to the constant nodes. In the final network all constant nodes are substituted by hardwiring to the systems $V_{cc}$ and $V_{dd}$ supplies.
• **Register**
  The register is used to store the signals on its data input port for a certain period, during which the stored value is continuously available at its output port. The register is of course a sequential module, with ports for the signals *clock* and *start*.

• **Communication-pads**
  These nodes represent asynchronic communication devices. These pads are sequential devices with an unknown delay. Thus, these devices have input ports for *start* and *clock* signals and one output port for the *ready* signal. There are three kinds of communication-pads:

  1. input-pad, with one output port through which data signals from the network’s environment are passed into the network.
  2. output-pad, with one input port through which data signals from the network are passed into the network’s environment.
  3. io-pad, with one inout port through which data signals can pass to or from the network and its environment. An io-pad may be used as an input or output-pad, to control the mode of the io-pad, another input port is available.

  The synchronization with the environment is performed completely by the pad. How the communication with the network’s environment is carried out is not important for us at all. The only thing we need to know is that a *start* should be given to the pad when some signals have to be *put* to or *get* from the environment. When the *start* has been given, it’s just a matter of waiting until the pad generates the signal *ready*. If that happens, we know that the communication has been executed successfully, i.e.

  1. the environment received our signals if we requested a put.
  2. the signals from the environment are available on the data output or inout port of the input- respectively io-pad if the requested operation was a get.

Like the data flow and control nodes, all nodes in a network are instantiated from a network graph. Thus, the user may create new network graphs and use the instantiations of these as network graphs in other network graphs. The modules represented by the nodes, instantiated from these self-made network graphs, are always sequential modules. Thus, a finished network graph always contains IO-nodes for the start, ready and clock signals. (see also section 4.3.4)

### 4.3.3 Interconnection nodes

The interconnection nodes represent network elements that are strongly related to transporting data (multiplexer, busses, tri-states). All interconnection modules are combinatoric modules.

• **Bus**
  A bus node has at least one input and at least one output port. The data transfers represented by all the edges connected to the same bus node, will be mapped on one bus in the final circuit.
• **Multiplexer**
  A multiplexer has one “control” input port, one “data” output port and at least two “data” output ports. The data output-ports are referred to by 0, 1, ... Multiplexers may be used to avoid signal conflicts, when two different output ports have to be connected to the same input port. The numerical value of the signal on the “control” port, specifies the name of the “data” input port which will be connected to the “data” output port.

• **Demultiplexer**
  A demultiplexer has one “control” input port, one “data” input port and at least two “data” input ports. The data input ports are referred to by 0, 1, ... The numerical representation of the signal on the “control” port, specifies which “data” output port is to be connected to the “data” input port. The other output ports will contain a high impedance.

• **Tri-state**
  A tri-state node has two output ports, one “control” input port and one “data” input port. If a signal high is set on the control port, the data input port is connected to the data output port. If the signal low is set on the control port, the output port will be high ohmic.

### 4.3.4 The controller

![Figure 4.2: A partitioning of the network graph](image)

A completed network graph always contains an input node for the start signal, an input node for the clock signal, an output node for the ready signal and a controller module to control the modules in the network. The start signal is required to make the controller transit to the first state. The ready signal will be passed to the environment from the controller through the “ready” output node of the network to indicate that the network has finished its job and that the signal on the output nodes are valid. These start and ready signals have been presented.
already in section 3.3. The clock will give the controller the notion of time. In figure 4.2 we see how the controller is related to the rest of the network.

4.3.5 Delay and timing of network modules

The delay of combinatoric devices must be less than a system cycle time, and execution may not cross a cycle transition. These are all the requirements for combinatoric devices. For sequential devices the delay and timing are more complicated. To make clear the timing of the sequential devices, we will discuss the timing of a device with known delay (multiplier) and a sequential device with an unknown delay (io-pad).

Let us discuss a multiplier with a delay of 2.1 cycles for both the operations multiplication and division. In figure 4.3 the timing diagram for this multiplier is presented. At each rising clock flank the signals on the data ports and control ports (signal \textit{mode} in diagram) are clocked into the multipliers internal buffers if \textit{start} $= 1$. At the end of $T_1$ the inputs are clock in by the multiplier, for multiplication. However, before the multiplier could finish multiplication, a \textit{start} signal is given at the end of $T_3$, such that the multiplier have to clock the input signals in its buffers and start multiplication again. Again, this multiplication will not finish. At the end of $T_4$, the input signals are clocked in, but this time for division instead of multiplication. Then after the delay (2.1 cycles) the output signals are valid and may be used by other devices in the network. At the end of $T_7$, multiplication is started and the result of that is presented 2.1 cycles later. The output data signals will be maintained by the multiplier as long as no \textit{start} reaches the multiplier at some rising clock flank. Finally another multiplication is started at the begin of $T_{13}$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{multiplier_timing.png}
\caption{Timing diagram for a multiplier with delay 2.1 cycles}
\end{figure}

Notice the resemblance between the diagrams of figure 4.1 and figure 4.3. Since the delay is unknown in figure 4.1, we only know that the output data signals can be clocked into other sequential devices at the end of $T_7$. Nothing is said about whether the output signals have been valid during a larger period in $T_7$. In figure 4.3, the delay of the multiplier is known, thus it is guaranteed that the output signals are valid during the last 0.9 cycle of $T_7$. Therefore, dur-
ing the last 0.9 cycles the output of the multiplier may be used by other devices in the network.

Devices with an unknown delay, must also have a delay specified. In this way, we know when the ready signals of these devices should be checked. If for instance, we know that a device will take at least 2.5 cycles to perform an operation, we could specify the delay as 3 cycles. The previous paragraph showed us that it is no use to have the estimated delay specified in fractions of a cycle for devices with unknown delay. In this way we will only have to check the ready signal of that device in the third cycle of execution till the cycle in which \textit{ready} becomes 1. The delay of a module will tell us whether we must check the ready signal. The ready signal should be checked when the delay of the device is a positive integer ($>0$). The ready signal has not to be checked if the delay is not an integer. One may think, what if there is a device with known delay and this delay is an integer, say 3. Well, in that case the delay should be specified as for instance 2.99 or 3.01. 2.99 when we are sure that the output of that device may be clocked into another sequential device at the end of the third state. If we cannot guarantee that, then we should specify the delay as 3.01.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{Timing diagram for an IO-pad}
\end{figure}

Let us discuss the IO-pad using the timing diagram in figure 4.4 of such a device. An IO-pad has got one input port for the signal \textit{mode}, which defines the mode of the device and one inout port for the \textit{data}. There are two modes for an IO-pad, the get ($\textit{mode} = 0$) and the put ($\textit{mode} = 1$) mode. According to the diagram, the IO-pad is in the put mode during the cycles $T_2$, $T_3$, $T_4$, $T_6$, $T_{10}$, $T_{14}$. In these cycles the inout port is configured as an input port. Thus, during these states the signal \textit{data} – \textit{out} in the diagram is undefined. Because \textit{data} – \textit{out} is defined according to the diagram in the cycles $T_0$ and $T_1$, we know that the IO-pad is in the get mode during these cycles. At the transition $T_1$ to $T_2$ the IO-pad comes into put-mode and the data on the inout port is clocked into the internal buffer at that transition. We see that it took the IO-pad two and a half cycle to put the data to the environment. At the transition $T_4$ to $T_5$ we wish to get some data from the environment. This time the communication is faster, at the transition $T_6$ to $T_8$ the data could be used by other devices. Because no \textit{start} is given at that transition, the same data will be available through the inout port during the whole cycle $T_6$. Then we have a sequence of one cycle communications: get, get, put, put.
CHAPTER 4. NETWORK GRAPH

This shows us that arbitrary communication through the IO-pad with the environment may be done in one cycle. Thus, it is clever to specify the delay of the IO pad as 1. Finally a get is done in 2 cycles and the result of the last put is not shown.

4.4 Network edge

Network edges represent connections between the ports of network nodes. The direction of the edge indicates which port is the source and which port the sink. Usually the network ports have a width larger than one. Therefore an edge usually represents a collection of wires between pins of the origin (source) port and the destination (sink) port. Attached to a network edge is a table which specifies how the pins are connected to each other by the edge.

![Network graph with two edges](image)

Figure 4.5: Two network edges in a network graph

For example, consider the two network edges in figure 4.5. Let the network ports of modules 1 and 2 have widths 3, and the port of module 3 has width 5. Let the connection tables for the edges be:

<table>
<thead>
<tr>
<th>connection table edge 1</th>
<th>connection table of edge 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin pin</td>
<td>destination pin</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

In figure 4.6 we have enlarged the situation of figure 4.5. Now we can see how the connections actually are at pinlevel.

Notice that the width of edge 1 is implicitly defined to be 2, by its connection table. In this way, the width of edge 2 is 4. Thus, the number of rows specified in the connection table equals the width of the edge. Notice that it is not allowed to have the same destination pin connected to more than one origin pin within one edge (signal conflict).

4.5 Some definitions

The definitions given on page 18 are of course also applicable to network graphs. But there are some definitions of operators, that are only applicable for the network objects.
CHAPTER 4. NETWORK GRAPH

Definition 4.1 (Controller in network graph)
Let \( nwg \) be a network graph containing one node representing the network controller. Then this node may be referred to as \( nwg.\text{ctrl} \).

Definition 4.2 (Combinatoric and sequential devices)
Let \( nwg \) be a network graph and \( nwn \) be a network node in that graph. And let \( nwn \neq nwg.\text{ctrl} \) if \( nwg \) contains a controller. Then the operators .comb and .sequ are defined on \( nwn \) as,

\[
\text{true} \quad \text{if } nwn \text{ represents a combinatoric device}\\
\text{false} \quad \text{otherwise}
\]

\[
\text{true} \quad \text{if } nwn \text{ represents a sequential device}\\
\text{false} \quad \text{otherwise}
\]

4.6 Validation of network graph

The design we get as input will contain a network graph that consists of a set of nodes, with no edges at all. Since our job is to add the edges and some other required modules to it, there is not much for us to verify in the network graph that we get as input.
Chapter 5

Graphlinks and Nodelinks

5.1 Introduction

The three graphs we have discussed in the previous chapters, are used to represent a design at each stage of the HLS process. In the NEAT system, a design is indicated by a graphlink between these graphs. A set of nodelinks is also a part of a graphlink and represent relations between the nodes in the data flow, network and control graphs. Scheduling and binding information for instance is stored in nodelinks.

5.2 Nodelinks

A nodelink is a combination of one network node, one data flow node and a set of control nodes. The network node represents the network module that is used to perform the operation represented by the data flow node. The set of control nodes represent the states in which the operation of the data flow node should be performed by the network node.

For each nodelink that is being used, it is required that there exists a graphlink in the database linked to,

1. the data flow graph from which the data flow node of the nodelink is instantiated and,
2. the network graph from which the network node of the nodelink is instantiated.

By defining the graphlinks for the standard data flow and network nodes in a library, we prescribe which type of data flow operator can be implemented by a network device. From such a graphlink we can easily extract the information about the purpose of each input, output and inout network node of the network graph linked to that graphlink. Thus, given some nodelink \( nl = (dfn, nwn, cts) \), where \( dfn \) is a data flow node, \( nwn \) is a network node and \( cts \) is a set of control nodes, we can find out for each network port \( port \in nwn \)

1. to which data flow port in \( dfn \) it corresponds or
2. for what kind of control signals it should be used.

\(^1\)In the NEAT system, there are many kinds of nodelinks. To keep it simple, we will stick with our description of nodelinks.
CHAPTER 5. GRAPHLINKS AND NODELINKS

If a network port is used for control purposes, then there are several possible signals, that could be connected to that network port:

1. **start**  
The start signal is required for all sequential devices. The port that is linked to this signal has width 1.

2. **ready**  
The ready signal is sometimes generated by sequential devices to indicate to the controller that the device has finished execution and is ready for the next operation. The port that is linked to this signal has width 1.

3. **clock**  
Network ports linked to this signal must be connected to the system clock of the network.

4. **dont-care**  
The port linked to this signal does not need to have some specific value.

5. **low**  
The port linked to this signal must receive the signal low on all its pins.

6. **high**  
The port linked to this signal must receive the signal high on all its pins.

7. **arbitrary bitvector**. For example 0-101-111. The width of the bitvector must be the same as the width of the port. The most right (left) bit is the least (most) significant bit. A 1 (0) stands for the signal high (low) on the corresponding pin.

5.3 Use and interpretation of nodeLinks

Nodelinks between the nodes of the graphs in a graphlink are used to indicate at which control nodes a data flow node is scheduled and to which network module it is bound. However, no exact rules have been given for the interpretation of nodelinks and not much is known about what is allowed and what is not allowed in using nodelinks. Especially when branch, merge, entry and exit nodes appear in the data flow graph, many interpretations of the nodelinks are possible. At this moment no designs are available, with control graphs containing conditional nodes. But it is expected that in the future, designs with condition nodes are delivered by the scheduler and binder. In this chapter we will show by examples how nodelinks can be used to represent many kinds of scheduling and bindings in designs including all the conditional nodes: branch, merge, entry and exit.

5.3.1 Plain graph

Figure 5.1 shows a scheduled and bound graphlink with a data flow graph without conditional nodes. In this example the multiplier has got a delay of 1.8 cycles and the adder has got a delay of 0.7 cycles. The nodelinks of the graphlink are presented in the form of a table in this figure. We see that the input nodes of the data flow graph are all implemented by input terminals and by the nodelinks nl4, nl5 and nl8, we see that the data through the input nodes are clocked into the registers reg1, reg2 and the multiplier at the begin of state S1. nl8 tells
CHAPTER 5. GRAPHLINKS AND NODELINKS

Figure 5.1: A GraphLink: a. Data flow graph b. Network graph c. Control graph d. NodeLinks

<table>
<thead>
<tr>
<th>Node Link</th>
<th>Data Flow Node</th>
<th>Network Node</th>
<th>Control Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>nl₁</td>
<td>input₁</td>
<td>in₁</td>
<td>S₀</td>
</tr>
<tr>
<td>nl₂</td>
<td>input₂</td>
<td>in₂</td>
<td>S₀</td>
</tr>
<tr>
<td>nl₃</td>
<td>input₃</td>
<td>in₃</td>
<td>S₀</td>
</tr>
<tr>
<td>nl₄</td>
<td>value₁</td>
<td>reg₁</td>
<td>S₁S₂S₃S₄</td>
</tr>
<tr>
<td>nl₅</td>
<td>value₂</td>
<td>reg₂</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl₆</td>
<td>value₃</td>
<td>reg₃</td>
<td>S₃</td>
</tr>
<tr>
<td>nl₇</td>
<td>value₄</td>
<td>reg₄</td>
<td>S₄</td>
</tr>
<tr>
<td>nl₈</td>
<td>*</td>
<td>multiplier</td>
<td>S₁S₂</td>
</tr>
<tr>
<td>nl₉</td>
<td>+1</td>
<td>adder</td>
<td>S₃</td>
</tr>
<tr>
<td>nl₁₀</td>
<td>+2</td>
<td>adder</td>
<td>S₄</td>
</tr>
<tr>
<td>nl₁₁</td>
<td>output</td>
<td>out</td>
<td>S₅</td>
</tr>
</tbody>
</table>
us that the multiplier is started in state $S_1$ and is also occupied in state $S_2$, this is necessary because the multiplier takes 1.8 cycles to perform the operation * . In state $S_3$ the operation $+1$ is executed by the adder, during $S_3$ the operands for $+1$-operation are delivered by the registers $reg_2$ and $reg_3$. From nodelink $nl_5$ one can see that $reg_2$ has clocked in the signals on its input at the start of $S_1$ and is occupied (frozen) during the states $S_2$ and $S_3$. Because the delay of the register is almost 0 cycles, the operation $value_3$ and $+1$ can be performed in the same state $S_3$. Then operation $+2$ is performed by the adder. Notice that the result of $+1$, calculated by the same adder, has to be stored in $reg_2 (nl_7)$ first. Finally the result of $+2$ is send out through the output node.

An important aspect of using the nodelinks is that the precedence relation in the data flow graph must be preserved in the network graph. For instance, if we add $S_2$ to the control nodes of $nl_6$, we could have an invalid schedule. Because then $reg_3$ will clock in the output of the multiplier, while the multiplier has not finished its calculations yet. However, using our delay index method (section 6.3) in interpreting nodelinks, even such cases can be handled correctly.

Another requirement is that we do not allow multicycling of combinatoric operations. This means that when a combinatoric device is used in a state, it must be ready with its execution when the end of that state is reached. This means also that we have restricted us to combinatoric devices with delays less than the cycle time of the system clock. For a combinatoric device to operate properly, it is needed that the input signals remain stable during the states in which the combinatoric device is scheduled. Notice, that if we have two adders, then $value_4$ is not required in the data flow graph.

From this example we have seen that a sequential device may be frozen when it has performed some operation, by linking a set of control nodes to it with more state nodes than it actually requires to execute. In this way we can also freeze combinatoric operations, but then it is required that the inputs of the combinatoric device are maintained during the freeze states also. An example of freezing combinatoric devices will be given in the next section.

### 5.3.2 BM-constructs in data flow graph

In figure 5.2 we see a graphlink with a data flow graph containing a bm-construct. The modules adder and compare, both have a delay of 0.6 cycles. All input signals are clocked into the registers at the begin of state $S_1$. Because the $>$ has to be executed before the $+$ and $-$ operators, the compare module is used in state $S_1 (nl_8)$. Because we do not allow multicycling, the $+$ and $-$ operator cannot be executed directly after the 0.6 cycles of the compare module. In state $S_2$ the $+$ or $-$ can then be carried out by the adder. Notice that the adder is used for two different purposes during the state $S_2$. This is possible because the $+$ and $-$ operator are performed under distinct conditions.

Notice that combinatoric operators may also be frozen. For example the operator $>$ is executed in state $S_1$ and frozen in the control nodes $S_2$ and $S_3$. Of course, a combinatoric device can only be frozen if its inputs are also frozen. In this case we see by nodelinks $nl_4$ and $nl_7$ that both inputs to the $>$-operator are also frozen during the states $S_2$ and $S_3$. From the nodelinks $nl_{12}$ and $nl_{13}$ we see that the branch and merge nodes linked to no network nodes.
CHAPTER 5. GRAPHLINKS AND NODELINKS

Notice that branch and merge data flow nodes are always handled as linked to combinatoric devices with delay 0, although they are not linked to any network device. To have the conditional nodes linked to some network node, we could decide to link it to some dummy network node $c$.

If we replace the adder and compare module in this example by a faster adder and compare-module, let's say 0.3 cycle delay, then we could perform the $>$ and $+$ or $-$ operator in one state. What happens is that $S_1$ is deleted from the control graph and the set of control nodes in the nodelinks and $S_2$ becomes the first state of the controller. Now, the data flow graph can be executed in one state. Then it is possible to calculate the condition ($>$) and perform the conditional body successively in one same state.

In figure 5.2 the comparator is frozen during the states $S_2$ and $S_3$. In this small example the compare-module is used only by the $>$-operator. But we could imagine that it is not always possible to freeze a network device during some arbitrary states, because the module might be needed to perform other operations during those states. In that case the output of the compare module has to be stored in another sequential device, which can be frozen during the required states. Usually a register is used for this purpose. Thus, in figure 5.2 we could put a data flow node $value_4$ between the output of the $>$ and the control ports of the conditional nodes. Then the nodelinks have to be updated:

<table>
<thead>
<tr>
<th>Node Link</th>
<th>Data Flow Node</th>
<th>Network Node</th>
<th>Control Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nl_4$</td>
<td>$value_1$</td>
<td>$reg_1$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$nl_7$</td>
<td>$constant$</td>
<td>$constant$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$nl_8$</td>
<td>$&gt;$</td>
<td>$compare$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$nl_{14}$</td>
<td>$value_4$</td>
<td>$reg_1$</td>
<td>$S_2 S_3$</td>
</tr>
</tbody>
</table>

The compare-module is now only occupied during the state $S_1$, while $value_1$ and $constant$ operators also occupy the network modules $reg_1$ and $constant$ during the state $S_1$ only. No new hardware has to be allocated, because $value_4$ can make use of $reg_1$.

When we replace the $+$-operation in figure 5.2 by a $*$-operation with 1.8 cycles delay and we do not wish to freeze the compare device during any state, then the graphlink in figure 5.3 shows us how a schedule with only three control state nodes may be represented. Because the multiplier is a sequential device, it has to be started in state $S_2$ if we wish to execute the whole data flow graph within 3 cycles. The control tokens to the branch nodes cannot be delivered by the $value_4$ node, because this node is mapped on a sequential device that has to be started at state $S_2$, thus the $+$-operation cannot be started in $S_2$ also. This problem is solved by connecting the control ports of the branch nodes to the $>$ operator, and using the node $value_4$ to indicate that the control value has to be stored during the states $S_2$, $S_3$ and $S_4$ (standby-state). Figure 5.3 is an example of an application of the special value node in bm-constructs as discussed in section 2.4.

5.3.3 BM-constructs in the control graph

In the previous examples, the control graph did not contain any conditional nodes. Now, we will show examples with control graphs containing BM-constructs. The advantage of using control graphs with conditional constructs is that the conditional bodies of one conditional construct do not have to last for the same number of cycles. For instance, in the previous
CHAPTER 5. GRAPHLINKS AND NODELINKS

Figure 5.2: A GraphLink: a. Data flow graph b. Network graph c. Control graph d. NodeLinks

<table>
<thead>
<tr>
<th>Node Link</th>
<th>Data Flow Node</th>
<th>Network Node</th>
<th>Control Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>nl1</td>
<td>input₁</td>
<td>in₁</td>
<td>S₀</td>
</tr>
<tr>
<td>nl2</td>
<td>input₂</td>
<td>in₂</td>
<td>S₀</td>
</tr>
<tr>
<td>nl3</td>
<td>input₃</td>
<td>in₃</td>
<td>S₀</td>
</tr>
<tr>
<td>nl4</td>
<td>value₁</td>
<td>reg₁</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl5</td>
<td>value₂</td>
<td>reg₂</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl6</td>
<td>value₃</td>
<td>reg₃</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl7</td>
<td>constant</td>
<td>constant</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl8</td>
<td>&gt;</td>
<td>compare</td>
<td>S₁S₂S₃</td>
</tr>
<tr>
<td>nl9</td>
<td>+</td>
<td>adder</td>
<td>S₂S₃</td>
</tr>
<tr>
<td>nl10</td>
<td>-</td>
<td>adder</td>
<td>S₂S₃</td>
</tr>
<tr>
<td>nl11</td>
<td>output</td>
<td>out</td>
<td>S₃</td>
</tr>
<tr>
<td>nl12</td>
<td>BR</td>
<td></td>
<td>S₂S₃</td>
</tr>
<tr>
<td>nl13</td>
<td>ME</td>
<td></td>
<td>S₂S₃</td>
</tr>
</tbody>
</table>
Figure 5.3: A GraphLink: a. Data flow graph b. Network graph c. Control graph d. NodeLinks
graphlink (figure 5.3) the body 1 of the bm-construct can be executed in one cycle, but because body 0 takes 2 cycles, the whole data flow graph has always to be executed in 3 cycles. In figure 5.4 a new control graph and new nodelinks are shown for the graphlink of figure 5.3. Using this control graph the data flow graph will be executed within 2 cycles, if body 1 is taken. Thus resulting in a better performance.

The branch nodes in the data flow graph are linked to the branch node in the control graph, to indicate that the branch node in the control graph is controlled by the >-operation. Because the data flow branch nodes are linked to a branch node in the control graph, the data flow merge node of the same construct has to be linked to the control merge node of the same construct as the control branch node (nl12 and nl13).

Notice that it is still necessary to store the output of >-node in a value node. This is required because in the state S4 (standby-state) we still need to know which conditional body had been executed, in order to connect the right device to the output device out. To indicate that value4 has to be implemented by reg1 during the whole ex-construct, nl14 contains the control nodes S2aS2bS3S4.

Notice that the registers for the value nodes value2 and value3 are frozen during the states S2b and S4. This means that we can do anything with the registers reg2 and reg3 during the states S2a and S3. We may do this because, when the states S2a and S3 are executed, we are sure that body 0 is being executed and then it is not necessary to have the values frozen in value2 and value3.
Figure 5.4: A GraphLink: a. Data flow graph b. Network graph c. Control graph d. NodeLinks
5.3.4 EX-constructs

Although we do not consider designs with loops in our interconnect and controller generator, we would like to show an example of a design with bm-constructs in the data flow as well as in the control graph. In figure 5.5 we see such a scheduled and bound design. In this example the multiplier has a delay of 1.7 cycle and the adder and compare modules have a delay of 0.7 cycles. Let us discuss some of the nodelinks.

nl1 The data flow node is always linked to a network input node and to the set of control nodes with only one element, the control input node.

nl2 At the begin of $S_1$ the data from the input node is clocked into the register $reg_1$. The content of that register is frozen during the state $S_2$ because its output is used in that state by the compare module to perform the $<$-operation.

nl3 The use of $value_2$ is based on the same idea as the node $value_4$ in figure 5.3: The control value has to be stored and we do not wish to delay the start of the $+$-operation. The value node has to be stored somewhere because in state $S_2$ we need to know which value ($value_1$ or $value_3$) must be compared by the $<$-operation. Therefore the control signals are clocked into $reg_2$ at the begin of $S_3$ and frozen during the states $S_4$ and $S_2$.

In the data flow graph initial tokens are put on the control ports of the entry nodes carrying a value corresponding to the 0 port. To take care of this in the hardware, at the transition of state $S_1$ to $S_2$, $value_2$ must be initialized by clocking a value corresponding to the 0 port of the ex-construction according to its selection list. The necessary hardware and connections will have to be arranged by our interconnector. Actually, according to the data flow graph semantics, $value_2$ should be initialized during all the states in the 0-body of the ex-construct. But this would result in occupation of a register during all the states in the 0-body, while the contents of that register are not being used anyway.

nl5 Notice that since $value_4$ is linked to the same register $reg_1$ there is a register transfer from $reg_1$ to $reg_1$ at the transition of $S_1$ to $S_2$. Of course it is not necessary to make this transfer in the hardware, during the interconnection we have to be aware of this.

nl6 The multiplier is frozen during the $S_2$ and $S_5$. Because when the loop ends, the output signals of the multiplier have to be connected to the output node in the network graph.

nl13 It should be obvious that in the way we deal with ex-constructs, each ex-construct results in an ex-construct in the control graph. And of course the data flow entry nodes are linked to the control entry node of the corresponding ex-construct.

nl14 Idem as nl13.
CHAPTER 5. GRAPHLINKS AND NODELINKS

Figure 5.5: A GraphLink: a. Data flow graph b. Network graph c. Control graph d. NodeLinks
5.4 Some definitions and restrictions

We have now discussed the major objects in a NEAT database, data flow graphs, network graphs, control graphs, graphlinks and nodelinks. In this section we will give some definitions and restrictions on the links in a NEAT database.

Definition 5.1 (NEAT database)
A NEAT database is a 4-tuple consisting of a set of data flow graphs, a set of network graphs, a set of control graphs and a set of graphlinks.

In this report we will restrict ourselves to one NEAT database at a time. We will call this database $db = (dfG, nwG, ctG, gL)$, with $dfG$ the set of data flow graphs, $nwG$ the set of network graphs, $ctG$ the set of control graphs and $gL$ the set of graphlinks.

Definition 5.2 (Graphlink)
A graphlink $gl \ingL$ is a 4-tuple consisting of a data flow graph in $dfG$, network graph in $nwG$, control graph in $ctG$ and a set of nodelinks.

The graphs in one graphlink represent together one design for the behavior given by the data flow graph. To indicate the state of the design, a few operators are defined.

Definition 5.3 (Some operators on a graphlink)
The operators .sched, .bound and .inter on a graphlink $gl$, are used to indicate the state in which the design represented by $gl$ is,

- $gl$.sched = \begin{cases} 
true & \text{if } gl \text{ is scheduled} \\
false & \text{otherwise} 
\end{cases}

- $gl$.bound = \begin{cases} 
true & \text{if } gl \text{ is module and register bound} \\
false & \text{otherwise} 
\end{cases}

- $gl$.inter = \begin{cases} 
true & \text{if } gl \text{ has been interconnected} \\
false & \text{otherwise} 
\end{cases}

Definition 5.4 (Delay of graphlink)
For each graphlink the delay is known, as the time (in cycles) that is needed for the network graph in that design to perform the operation given by the data flow graph. The delay of a graphlink $gl$ is denoted by $gl$.delay.

The nodelinks in a graphlink are used to store information about relations between the three graphs. In our case this information concerns the result of the binding and scheduling processes.
Definition 5.5 (Nodelink)
Let there be a graphlink $gl = (dfg, nwg, ctg, nL)$. $dfg$ is the data flow graph, $nwg$ the network graph and $ctg$ the control graph in $gl$. And $nL$ is the set of nodelinks in $gl$. Then a nodelink $nl$ in $nL$ is a 3-tuple consisting of a data flow node in $dfg$, a network node in $nwg$ and a set of control nodes in $ctg$. Thus,

$$nl \in nL \subseteq dfg.Nodes \times nwg.Nodes \times 2^{ctg.Nodes}$$

For each nodelink $nl = (dfn, nwn, cts)$ the operators $df$, $nw$ and $ct$ are defined as

$$nl.df = dfn$$

$$nl.nw = nwn$$

$$nl.ct = cts$$

 Definition 5.6 (Data flow to network port mapping)
Let there be a nodelink $nl = (dfn, nwn, cts)$. Then the portmapping $dfport$ is a function,

$$nl.dfport : dfn.Ports \rightarrow nwn.Ports \cup \{\epsilon\}$$

that returns the network port corresponding to the data flow port in its argument. If the data flow port in the argument does not match any network port in $nwn$ then $\epsilon$ is returned.

 Definition 5.7 (Network to data flow port mapping)
Let there be a nodelink $nl = (dfn, nwn, cts)$. Then the portmapping $nwport$ is a function on $nl$,

$$nl.nwport : nwn.Ports \rightarrow dfn.Ports \cup \{\epsilon\}$$

that returns the data flow port corresponding to the network port in its argument. If the network port in the argument does not match any data flow port in $nwn$ then $\epsilon$ is returned.

We restrict the use of nodelinks $nl = (dfn, nwn, cts)$ in scheduled and bound graphlinks with the following constraints:

1. if $dfn$ is a non conditional node:
   $$(\exists! : gl' \in g.Ls \land gl' = (dfg', nwg', ctg', nL') \land gl'.sched \land gl'.bound \land gl'.inter : dfn.instedFrom = dfg' \land nwn.instedFrom = nwg')$$
i.e. For each nodelink there must be a corresponding graphlink in the database containing the information considering port and signal mappings.

2. $$(\forall : dfn \in dfg : (\exists! : nl \in nL : nl.df = dfn))$$
i.e. All data flow nodes are linked to exactly one nodelink in $gl$.

Because for each nodelink there always corresponds one graphlink, we may also talk about the delay of a nodelink.
Definition 5.8 (Delay of nodelink)
The delay of a nodelink $nl = (dfn, nwn, cts)$, denoted by $nl\text{.delay}$, is defined as,

$$nl\text{.delay} = gl\text{.delay}$$

with $dfn\text{.insteadFrom} = gl\text{.df}$ and $nwn\text{.insteadFrom} = gl\text{.nw}$. 

In other words $nl\text{.delay}$ gives us the delay required by the network device in $nl$ to perform the operation given by the data flow node in $nl$. Because each data flow node $dfn$ is linked to exactly one nodelink, we may also define the delay of a data flow node.

Definition 5.9 (Delay of data flow node)
Let $gl = (dfg, nwg, ctg, nL)$ be a graphlink with the data flow graph $dfg$, network graph $nwg$, control graph $ctg$ and the set of nodelinks $nL$, such that $gl\text{.sched} \land gl\text{.bound}$. Then the delay of a data flow node $dfn$ in $dfg$, denoted by $dfn\text{.delay}$, is defined as,

$$dfn\text{.delay} = nl\text{.delay}$$

with $nl \in nL \land nl\text{.df} = dfn$. 

\[\Box\]
Chapter 6

Modeling the use of nodelinks

6.1 Introduction

In this chapter we will present our own model for using nodelinks. We will do this for all designs in which the data flow and control graphs do not contain entry and exit nodes. The reason for this is because loops in the data flow and control graphs makes things much more complicated. Since at this moment no designs with loops can be handled by the scheduler or binder, we focus ourselves on designs without loops. We will start with a discussion about conditions in data flow and control graphs. In section 6.2, algorithms are presented to calculate the conditions of nodes and edges in graphs. Based on these conditions, we will introduce the delay index in section 6.3. With the delay index we will be able to formulate the restrictions on the use of nodelinks to store scheduling and binding information.

6.2 Condition analysis on data flow and control graphs

In this section we will present a method to calculate the data flow conditions of edges and nodes in data flow and control graphs. The method will be used for designs with bm constructs in the data flow graph as well as in the control graph. Only the bm construct without the special value node will be considered. Thus, all control ports of the conditional nodes of one conditional construct is connected to the same port.

The conditions of edges and nodes in the data and control graphs will give us the information on when a node could be executed and token could flow through an edge. After defining a few often used terms, we will discuss the data flow conditions of the data flow edges by examining how data flow nodes execute. Then we will show an algorithm by which the conditions can be calculated. A similar algorithm is used for determining the data flow conditions in the control graph.

Definition 6.1 (Data flow and control paths)
A data flow (control) path $p$ is a sequence of data flow (control) edges $(e_1, e_2, \ldots, e_n)$ with the following properties:

1. The origin port of $e_1$, also called the origin of the path, is an output port, but not owned by a conditional node.
2. The destination port of \( e_n \), also called the destination of the path, is an input port, but not a data input port of a conditional node.

3. The destination port of \( e_i \) is a data input port of a conditional node and the origin port of \( e_{i+1} \) is a data output port of the same node, \((0 < i < n)\).

Definition 6.2 (Paths in data flow and control graphs)
Let \( g \) be a data flow or control graph. Then the set of paths in that graph is denoted by \( g.Paths \).

Definition 6.3 (Origin and destination of a path)
Let \( p = (e_1, e_2, \ldots, e_n) \) be a data flow or control path. Then the \( .orig, .dest, .origport \) and \( .destport \) operators are defined on \( p \) as,

\[
\begin{align*}
p.orig &= e_1.orig \\
p.dest &= e_n.dest \\
p.origport &= e_1.origport \\
p.destport &= e_n.destport
\end{align*}
\]

Definition 6.4 (Condition generating port)
Let there be a data flow graph \( dfg \). Let \( port \) be a port of a node in \( dfg \). Then \( port \) is a condition generating port if and only if

\[
(\exists: p \in dfg.Paths \land p.dest.type = "branch"|"merge"|"entry"|"exit": p.origport = port)
\]

Definition 6.5 (Condition generating ports in data flow graph)
Let there be a data flow graph \( dfg \). Then the set of all condition generating ports in this graph is denoted by \( dfg.condPorts \).

Each port of the nodes in the data flow graph contains a set of pins (definition 2.13). We will define the data flow condition as a boolean expression, in which each pin (bit) of a condition generating port corresponds to a literal (boolean variable) in the expression.

Definition 6.6 (Literal function)
Let there be a data flow graph \( dfg \). Then the literal (boolean variable) function \( dfg.lit \) is a bijection from the pins of condition generating ports in \( dfg \) onto a set of boolean variables \( B \).

\[
dfg.lit : (\cup: port \in dfg.condPorts : port.Pins) \rightarrow B
\]
**Definition 6.7** (Data flow condition)
Let there be a data flow graph \( dfg \) and let \( B \) be the range of its literal function \( dfg.lit. \) Then a data flow condition is a boolean expression, in which the literals are elements of \( B. \)

**Definition 6.8** Data flow condition of a data flow or control edge
Let there be a graphlink \( gl = (dfg, nwg, ctg, nL) \). Let \( e \) be an edge and \( e \in dfg.Edges \) or \( e \in ctg.Edges. \) Then the data flow condition of edge \( e \), denoted by \( C_e \), contains the information about what values should be carried by the tokens generated by the condition generating ports, so that token transport through the edge \( e \) is possible. Thus,

\[ C_e \Rightarrow \text{token transport through } e \text{ is possible} \]

**Definition 6.9** (Data flow condition of a data flow or control node)
Let there be a graphlink \( gl = (dfg, nwg, ctg, nL) \). Let \( n \) be a node and \( n \in dfg.Nodes \) or \( n \in ctg.Nodes. \) Then the data flow condition of \( n \), denoted by \( C_n \), contains the information about what values should be carried by the tokens generated by the condition generating ports in \( dfg, \) so that node \( n \) can be executed. Thus,

\[ C_n \Rightarrow \text{execution of node } n \text{ is possible} \]

**Definition 6.10** (Data flow condition of a data flow or control path)
Let \( p = (e_1, \ldots, e_n) \) be a data flow or control path. Then the data flow condition of this path is the product of the data flow conditions of the edges in \( p: \)

\[ C_p = \bigwedge_{i=1}^{n} C_{e_i} \]

**Definition 6.11** (Type of a data flow path)
Let \( p = (e_1, \ldots, e_n) \) be a data flow path. Then the type of \( p \), denoted by \( p\.type \) is defined as\(^1\),

\[ p\.type = \begin{cases} 
\text{"data"} & \text{if } (\forall : 0 < i \leq n : e_i\.type = \text{"data"}) \\
\text{"source"} & \text{otherwise}
\end{cases} \]

\( ^1 \text{Remember that control edges are also data edges (see section 2.3).} \)

6.2.1 Relations between data flow conditions of data flow edges.
In this section we will examine and present the relations between the conditions of the data flow edges. It is easy to understand that when the data flow graph is a plain graph, all conditions of the edges should be \( \text{true} \), simply because there are no condition generating ports in the graph. By using the definition of how to execute a plain node according to
the token formalism, we get the following relation between the data flow conditions of edges connected to a plain node (see figure 6.1a):

$$ C_{\text{out}} = \bigwedge_{i=0}^{n-1} C_{\text{in},i} $$

(6.1)

Here, $C_{\text{out}}$ is the data flow condition of an arbitrary output edge. Since a plain node will only put a token on its outputs when there are tokens on all its inputs, it is obvious that the condition to have a token on all the output edges is the product of all the input-edge-conditions. In the special case that a node do not have any input ports we get:

$$ C_{\text{out}} = \text{true} $$

(6.2)

So, the edges connected to an input node may always carry tokens, independent on what values the condition generating ports produce.

Now, let us examine the relation of the conditions of edges connected to conditional nodes.

**Edges connected to branch nodes**

To execute a branch we need to put a token on its data input port and a token on its control port, carrying the selection-value. According to this value the token on the data input port will be passed to one of the output ports. Having this in mind we come to the relation between the data flow conditions of edges connected to a branch $\text{dfn}$:

$$ C_{\text{out},i} = C_{\text{in}} \land C_{\text{sel},i} \quad (0 \leq i < n) $$

(6.3)

With the notation of edges according to figure 6.1b. $C_{\text{sel},i}$ is the data flow condition under which the $i^{th}$ output port of the branch is selected.

In figure 2.1 we saw a branch node $\text{dfn}$ directly controlled by a plain node ($>\text{)$. Let us call the output port of the $>\text{-node in that graph, port. Then the data flow condition for the selection of port $i$ of the branch node $\text{dfn}$ is given by

$$ C_{\text{sel},i} = C_{\text{ctrl}} \land (\text{dfn.sel(port)} = i) \quad (0 \leq i < n) $$

(6.4)

$\text{dfn.sel}$ stands for an evaluation function that returns the name of the data input port that is selected according to the selection function applied on the token generated by port $\text{port}$.
Definition 6.12 (Selection function of branch node)
Let there be a data flow graph $dfg$ and a branch node $dfn \in dfg.Nodes$. Let $port$ be a port and $port \in dfg.condPorts$. Then the selection function of the node $dfn$ is defined as

$$
dfn.sel : dfg.condPorts \rightarrow (\cup : p \in dfn.outPorts : \{p.name\})
$$

$$
dfn.sel(port) = i
$$

with $i$ the name of the port, that would have been selected according to the selection list of $dfn$ if the token generated by $port$ arrived at the control port of $dfn$.

Notice that "$dfn.sel(port) = i$" in equation 6.4 gives us the data flow condition to have port $i$ selected if the token generated by $port$ arrives at the control port of $dfn$.

Example 6.1 (Selection condition)
Let $dfg$ be the data flow graph in figure 2.1. Let $port$ be the output port of the $>$-node in $dfg$. Let $dfn$ be one of the branch nodes in $dfg$. Let $dfe$ be the edge between $port$ and the control port of $dfn$. Then the selection condition for the 0 body in the bm construct is given by

$$(C_{dfe} \land (dfn.sel(port) = 0)) \equiv dfg.lit(port.0)$$

and the selection condition for the 1 body is given by

$$(C_{dfe} \land (dfn.sel(port) = 1)) \equiv dfg.lit(port.0)$$

Next, consider the situation as pictured in figure 2.3. The control ports of the last bm construct are now connected to several data flow paths. More than one condition generating port is able to deliver the control tokens for the last bm construct. Depending on the conditions of the data flow paths to the control ports of the last bm construct, only one condition generating port will provide the control token, carrying the selection value. To meet the needs for these "conditional condition" constructs we have to modify relation 6.4:

$$C_{sel,i} = \bigvee_{p \in P} C_p \land (dfn.sel(p.origin) = i) \quad (0 \leq i < n - 1) \quad (6.5)$$

Here, $P$ is the set of all data flow paths to the control port of the branch node.

Example 6.2 (Selection condition with conditional condition)
Let $dfg$ be the data flow graph in figure 2.3. Let $dfn$ be a branch node of the last bm construct in $dfg$. Let $port_0$ be the output port of the input node $(c)$ in $dfg$. Let $port_1$ be the output node of the constant1 node in $dfg$. Let the output port of the $=!$ node in $dfg$ be $port_2$. Let $p_1$ be the path from $port_0$ to the control port of $dfn$. Let $p_2$ be the path from $port_1$ to the control port of $dfn$. Then the selection condition for the 0 body in the last bm construct is given by

$$C_{sel,0} = \bigvee_{p \in \{p_1,p_2\}} C_p \land (dfn.sel(p.origin) = 0) = $$

$$\overline{dfg.lit(port_2.0)} \land \overline{dfg.lit(port_0.0)} \lor dfg.lit(port_2.0) \land \overline{dfg.lit(port_1.0)}$$
and the selection condition for the 1 body is given by
\[
C_{sel,0} = \bigvee_{p \in \{p_1, p_2\}} C_p \land (dfn.sel(p\text{-}origin) = 0) = \\
\overline{dfg\text{-}lit(port_2\text{-}0)} \land dfg\text{-}lit(port_0\text{-}0) \lor dfg\text{-}lit(port_2\text{-}0) \land dfg\text{-}lit(port_1\text{-}0)
\]

Substitution of 6.5 into 6.3 gives us the final equation:
\[
C_{out,i} = C_{ein,i} \land \left[ \bigvee_{p \in P} C_p \land (p\text{-}dest.sel(p\text{-}origin) = i) \right] \quad 0 \leq i < n - 1 
\] (6.6)

**Edges connected to merge nodes**

A merge node will pass the token on one of its data input ports to the output, according to the value carried by the token on the control port. This results in the relation:
\[
C_{out} = \bigvee_{i=0}^{n-1} C_{in,i} \land C_{sel,i} 
\] (6.7)

In this equation the edges are named according to figure 6.1c. $C_{sel,i}$ is the data flow condition under which the $i^{th}$ input port of the merge is selected. Since merge nodes are only allowed to appear in valid branch merge constructs (without special value nodes), we know that the data flow conditions of the input edges of the merge node satisfy:
\[
C_{in,i} \Rightarrow C_{sel,i} \quad (0 \leq i < n) 
\] (6.8)

This means that in equation 6.7 the condition $C_{in,i}$ already contains the condition $C_{sel,i}$. Thus, equation 6.7 reduces to:
\[
C_{out} = \bigvee_{i=0}^{n-1} C_{in,i} 
\] (6.9)

Using these relations it is possible to calculate the data flow conditions of all the edges in the data flow graph. In the next section we will present an algorithm for this job.

**6.2.2 Calculation of data flow conditions in data flow graph.**

The algorithm `DfGraph::setDfConditions` will pick an arbitrary edge in the data flow graph, and calculate its condition by `DfEdge::setDfCondition`. `DfEdge::setDfCondition` will do this by having first calculated the conditions of all the edges that leads to the origin node of the edge. Using the relations discussed in the previous section (equations 6.1, 6.6 and 6.9), the condition of the actual edge is determined, depending on whether the node is a branch, merge or plain node.

The data flow conditions of the data flow nodes can easily be determined when the conditions of the edges are known. If the node is not a branch node then the conditions of all its output edges are the same. The condition of the node is in that case the same as the condition of its output edges. If the node is a branch node then its condition is the union of the conditions of all its output edges.
Algorithm 6.1 (Set the conditions of all edges and nodes in data graph)

DfGraph::setDfConditions()
{
   /* Pre: this = dfg */
   for all (dfe ∈ dfg)
   {
      dfe.cond = false
   }
   end all
   for all (dfe ∈ dfg)
   {
      dfe.setDfCondition()
   }
   end all
   for all (dfn ∈ dfg.Nodes)
   {
      dfn.cond = false
      dfn.setDfCondition()
   }
   end all
}
Algorithm 6.2 (Set all conditions of the data flow edges)

DfEdge::setDfCondition()
{
    /* Pre: this = dfe ∈ dfg.Edges */
    if (dfe.cond ≠ false) return
    for all (port ∈ dfe.orig.inPorts ∧ e ∈ port.Edges)
    {
        e.setDfCondition()
    }
    end all
    if (dfe.orig.type = "branch")
    {
        for all (p ∈ dfg.Paths ∧ p.destport = dfe.orig.port("control"))
        {
            dfe.cond = dfe.cond ∨ (p.cond ∧ p.dest.sel(p.origport) = dfe.origport.name)
        }
        end all
        dfe.cond = dfe.cond ∧ dfe.orig.port("in").edge.cond
    }
    else if (dfe.orig.type = "merge")
    {
        for all (port ∈ dfe.orig.inPorts ∧ port.name ≠ "control" ∧ e ∈ port.Edges)
        {
            dfe.cond = dfe.cond ∨ e.cond
        }
        end all
    }
    else
    {
        dfe.cond = true
        for all (port ∈ dfe.orig.inPorts ∧ port.name)
        {
            dfe.cond = dfe.cond ∧ port.edge.cond
        }
        end all
    }
}
Algorithm 6.3 (Set condition of the data flow node)

DfNode::setDfCondition()
{
    /* Pre: this = dfn */
    for all (port ∈ dfn.outPorts ∧ e ∈ port.Edges)
    {
        dfn.cond = dfn.cond ∨ e.cond
    }
    end all
}
6.2.3 Calculation of data flow conditions in the control graph.

The same data flow condition analysis as we have conducted on the data flow graph, can be applied to the control graph. This is possible because the control graph is, like the data flow graph more or less based on the same token flow principle. Except that all the control ports of conditional nodes in a control graph are virtually connected to a data flow condition generating port. Thus, we will keep talking about data flow conditions, even if the condition is applied on a control edge. In the previous chapter we saw that the information about the virtual connections between condition generating ports in the data flow graph and the control ports of the conditional nodes in the control graph, were stored in the nodelinks also. However, using the delay index concept, we will use the nodelinks solely for scheduling and binding information. And a conditional node in the data flow graph could be linked to a set of control nodes containing conditional nodes of different condition constructs. Thus, it will be very hard to see the virtual connections through these nodelinks.

To keep things simple we propose another approach for storing the information about the virtual connections. We will keep in each conditional node in the data and control graph a condition construct number. The conditional nodes that form one construct in the data flow or control graphs will have the same number. Each construction will have its own unique condition construct number within each graph. So, by looking at the condition construct number of a conditional node, we know to which construct it belongs. Now, the virtual connection between the conditional nodes in the control graph and those in the data flow graph is stored by assigning the same condition construct number to corresponding conditional constructs in the data flow and control graphs.

**Definition 6.13** (Condition number of condition construct node)

Let \( n \) be a conditional node. Then the condition construct number of this node in an integer denoted by \( n.ccnum \).

The algorithms 6.4, 6.5 and 6.6 are used to calculate the data flow conditions of the control nodes and edges. The algorithms are very similar to those used for the data flow graph. Notice the difference between the iteration argument of the second loops in `DfEdge::setDfCondition` and `CtEdge::setDfCondition`. In `CtEdge::setDfCondition` we have to consider the paths to the control ports in the data flow graph, of branch nodes with the same condition construct number.
Algorithm 6.4 (Set the conditions of all edges and nodes in control graph)

CtGraph::setDfConditions()
{
    /* Pre: this = ctg */
    for all (cte \in ctg)
    {
        cte.cond = false
    }
    end all
    for all (cte \in ctg)
    {
        cte.setDfCondition()
    }
    end all
    for all (ctn \in ctg.Nodes)
    {
        ctn.cond = false
        ctn.setDfCondition()
    }
    end all
}
Algorithm 6.5 (Set all conditions of the control edges)

CtEdge::setDfCondition()
{
    /* Pre: this = cte and dfg is the data flow graph in the design */
    if (cte.cond \neq false) return
    for all (port \in cte.orig.inPorts \land e \in port.Edges)
    {
        e.setDfCondition()
    }
    end all
    if (cte.orig.type = "branch")
    {
        for all (p \in dfg.Paths \land p.dest.type = "branch" \land p.dest.ccnum = cte.orig.ccnum)
        {
            cte.cond = cte.cond \lor (p.cond \land p.dest.sel(p.origport) = cte.origport.name)
        }
        end all
        cte.cond = cte.cond \land cte.orig.port("in").edge.cond
    }
    else if (cte.orig.type = "merge")
    {
        for all (port \in cte.orig.inPorts \land port.name \neq "control" \land e \in port.Edges)
        {
            cte.cond = cte.cond \land e.cond
        }
        end all
    }
    else
    {
        cte.cond = true
        for all (port \in cte.orig.inPorts \land port.name)
        {
            cte.cond = cte.cond \land port.edge.cond
        }
        end all
    }
}
Algorithm 6.6 (Set condition of the data flow node)

```cpp
CtNode::setDfCondition()
{
    /* Pre: this = ctn */
    for all (port ∈ ctn.outPorts ∧ e ∈ port.Edges)
    {
        ctn.cond = ctn.cond ∨ e.cond
    }
    end all
}
```
6.3 Delay index

In this section we will present the delay index of data flow nodes. This is a very useful instrument in reasoning about the use and interpretation of nodelinks. Before we define the delay index we need to define the delay of a control node.

**Definition 6.14 (Delay of control node)**

Let \( n \) be a control node. The delay of \( n \), denoted by \( n.\text{delay} \) is defined as,

\[
   n.\text{delay} = \begin{cases} 
   0 & \text{if } \text{ctn.type} = \text{"branch"} \text{ or } \text{"merge"} \\
   1 & \text{otherwise}
   \end{cases}
\]

The concept of the delay index is based on regarding all the control nodes in the control graph as equal nodes. The delay of the control nodes will make the distinction between the conditional nodes and the other nodes in the control graph.

**Definition 6.15 (Delay index)**

Let there be a bound and scheduled graphlink \( gl = (dfg, nwg, ctg, nL) \). Let there be a nodelink \( nl = (dfn, nwn, cts) \in nL \). Then the delay index of data flow node \( dfn \) during the control node \( ctn \in ctg.\text{Nodes} \) is defined by the delay index function

\[
   I : dfg.\text{Nodes} \times ctg.\text{Nodes} \rightarrow \mathbb{R}^+
\]

\[
   I(dfn, ctn) = \begin{cases} 
   \infty & \text{if } ctn \notin cts \land (dfn.\text{cond} \land ctn.\text{cond} = \text{false}) \\
   I_s(dfn, ctn) & \text{if } ctn \in cts \land (dfn.\text{cond} \land ctn.\text{cond} \neq \text{false}) \land nwn.\text{seq}
   \end{cases}
\]

with \( I_s \) and \( I_c \) the delay index functions for respectively sequential and combinatoric devices.

The delay index function \( I(dfn, ctn) \) returns for a data flow node \( dfn \) the delay (in cycles) from the start of the control node \( ctn \), required by the network device to perform the operation, under the condition \( dfn.\text{cond} \land ctn.\text{cond} \). If \( ctn \) is not contained in the set of control nodes \( (cts) \) linked to \( dfn \), then \( dfn \) is definitely not scheduled at \( ctn \) and \( \infty \) is returned. We will see that the delay index functions of the sequential and combinatoric devices are quite different.

**Definition 6.16 (Delay index for sequential devices)**

Let there be a bound and scheduled graphlink \( gl = (dfg, nwg, ctg, nL) \). Let there be a nodelink \( nl = (dfn, nwn, cts) \in nL \), with \( nwn \) a sequential device. Then the delay index of \( dfn \) during the control node \( ctn \in cts \) is defined by the delay index function for data flow nodes mapped on sequential devices,

\[
   I_s : dfg.\text{Nodes} \times ctg.\text{Nodes} \rightarrow \mathbb{R}^+
\]

\[
   I_s(dfn, ctn) = \begin{cases} 
   I_s1(dfn, ctn) & \text{if } I_s1(dfn, ctn) \neq \infty \\
   I_s2(dfn, ctn) & \text{if } I_s1(dfn, ctn) = \infty
   \end{cases}
\]

with \( I_{s1} \) and \( I_{s2} \) defined as,

\[
   I_{s1}, I_{s2} : dfg.\text{Nodes} \times ctg.\text{Nodes} \rightarrow \mathbb{R}^+
\]
**CHAPTER 6. MODELING THE USE OF NODELINKS**

\[ I_{s1}(dfn, ctn) = \begin{cases} 
0 & \text{if } x \leq 0 \\
x & \text{if } x > 0 
\end{cases} \]

\[ I_{s2}(dfn, ctn) = \begin{cases} 
nl.delays & \text{if } (\forall : dfn' \in \text{pred}(dfn) \land ctn' \in \text{pred}(ctn) \\
& : I(dfn', ctn') \leq ctn'.delay) \\
\infty & \text{otherwise} 
\end{cases} \]

with \( x = (\max : ctn' \in \text{pred}(ctn) \land (ctn'.cond \land dfn.cond) \neq \text{false} : I(dfn, ctn') - ctn'.delay) \).

\( I_s(dfn, ctn) \) is determined by first examining whether \( dfn \) has been scheduled in the predecessors of \( ctn \). \( I_{s1}(dfn, ctn) \) will be a number \( \neq \infty \), if \( dfn \) has been started in a earlier state then \( ctn \). If \( I_{s1}(dfn, ctn) \) equals \( \infty \), then we have to determine whether it is possible to have \( dfn \) executed starting from \( ctn \). In which case \( I_{s2}(dfn, ctn) \) returns the delay needed to execute \( dfn \).

\( I_{s1}(dfn, ctn) \) is determined by first looking at whether \( dfn \) has been scheduled also in the direct predecessor of \( ctn \). Let us first consider the case when \( ctn \) has got exactly one predecessor. \( I_{s1} \) is then determined as the delay index of \( dfn \) in the preceding control node \( (ctn') \), minus the delay consumed by that control node. Now, \( x \) will be less then zero, if the execution of the operation \( dfn \) already has finished or is finishing in \( ctn' \). \( I_{s1} \) equals zero in that case, indicating that the execution of \( dfn \) has finished before the end of \( ctn' \), i.e. the begin of \( ctn \). If \( x \) is positive, then the execution still needs \( I_{s1} \) cycles to finish its execution (counting from the begin of \( ctn \)). Thus, \( I_{s1} \) is returned in this case.

Let us now consider the calculation of \( I_{s1} \) when \( ctn \) has more than one predecessor. Then \( ctn \) must be a node of type "merge". We will then examine the delay index for the predecessors one by one. This results in as many delay indexes as there are predecessors. However, we only have to look at the relevant predecessors \( (ctn'.cond \land dfn.cond) \neq \text{false} \). From these delay indexes we will take the maximum. In this way it is guaranteed that \( dfn \) still need at most \( I_{s1} \) cycles, before it is ready.

At this point one might wonder why we only look at the relevant predecessors \( (ctn'.cond \land dfn.cond) \neq \text{false} \) in determining \( I_{s1} \). Well, the weakest condition to have \( dfn \) scheduled in \( ctn \) is given by \( (ctn.cond \land dfn.cond) \). Thus, we only have to examine the predecessors \( ctn' \)'s of \( ctn \) that are relevant \( (ctn'.cond \land dfn.cond) \neq \text{false} \) in order to guarantee that if \( (ctn.cond \land dfn.cond) \) then it takes at most \( I_{s1} \) cycles for \( dfn \) to finish execution. Notice that we have examined the complete condition space covered by \( (ctn.cond \land dfn.cond) \), because

\[ (ctn.cond \land dfn.cond) \equiv (\forall : ctn' \in \text{pred}(ctn) : (ctn'.cond \land dfn.cond)) \]

If \( I_{s1} \) is equal to \( \infty \), then we know that it is not guaranteed that \( dfn \) is being executed already during \( ctn \)'s predecessors. So, we have to find out whether \( dfn \) starts its execution during the control node \( ctn \). The sequential device performing \( dfn \) may start execution in \( ctn \), only if the predecessors of \( dfn \) already finished their execution during the predecessors of \( ctn \). If that is the case, \( I_{s2} \) will return the delay for performing the operation specified by \( dfn \), otherwise \( \infty \) is returned.
6.17 (Delay index for combinatoric devices)
Let there be a bound and scheduled graphlink $gl = (dfg, nwg, ctg, nL)$. Let there be a nodelink $nl = (dfn, nwn, cts)$ in $nL$, with $nwn$ a combinatoric device. Then the delay index of $dfn$ during the control node $ctn \in cts$ is defined by the delay index function for combinatoric data flow nodes,

\[
I_c : dfg.Nodes \times ctg.Nodes \rightarrow \mathbb{R}^+
\]

\[
I_c(dfn, ctn) = \begin{cases} 
I_{c1}(dfn, ctn) & \text{if } I_{c1}(dfn, ctn) \neq \infty \\
I_{c2}(dfn, ctn) & \text{if } I_{c1}(dfn, ctn) = \infty 
\end{cases}
\]

with $I_{c1}$ and $I_{c2}$ defined as,

\[
I_{c1}, I_{c2} : dfg.Nodes \times ctg.Nodes \rightarrow \mathbb{R}^+
\]

\[
I_{c1}(dfn, ctn) = \begin{cases} 
0 & \text{if } I(dfn, ctn') - ctn'.delay < 0 \land 
(V : dfn' \in pred(dfn) : I(dfn', ctn) = 0) \\
\infty & \text{otherwise}
\end{cases}
\]

\[
I_{c2}(dfn, ctn) = \begin{cases} 
x & \text{if } x < ctn.delay \\
\infty & \text{otherwise}
\end{cases}
\]

with $x = (\max : dfn' \in pred(dfn) : I(dfn', ctn)) + dfn.delay$.

$I_c(dfn, ctn)$ is determined by first examining whether $dfn$ has been scheduled in the predecessors of $ctn$. $I_{c1}(dfn, ctn)$ will be zero, if $dfn$ has been started in a earlier state then $ctn$. If $I_{c1}(dfn, ctn)$ equals $\infty$, then we have to determine whether it is possible to have $dfn$ executed in $ctn$. In which case $I_{c2}(dfn, ctn)$ returns the cycles required for $dfn$ to finish execution, starting from the beginning of $ctn$.

$I_{c1}(dfn, ctn)$ equals zero if $dfn$ already finished execution during previous control nodes and the predecessors of $dfn$ are still active during $ctn$. Thinking in terms of network modules this means that the performance of the operation represented by $dfn$ must have been done in an earlier state then $ctn$. And the inputs to the network module representing $dfn$ must be maintained till and during $ctn$. Again we only examine the relevant predecessor states of $ctn ((ctn'.cond \land dfn.cond) \neq false)$.

If $I_{c1}$ equals $\infty$, then we have to check whether $dfn$ could be executed in $ctn$. $dfn$ can be executed in $ctn$ only if the predecessors of $dfn$ finishes in $ctn$ and there is enough time left for $dfn$ to finish in that control node. If $dfn$ could finish within the end of $ctn$ then the needed cycles is given by $I_{c2}$. If $dfn$ could not finish before $ctn$ ends, $I_{c2}$ will be equal to $\infty$, because we do not allow multicycling of combinatoric devices.

6.4 Rules for using nodelinks

By using the delay index function we can now give the rules for using nodelinks. Let there be a bound and scheduled graphlink $gl = (dfg, nwg, ctg, nL)$, then
1. \((\forall \: nl \in nL \land nl.df.type = \text{"input"} : nl.ct = \{ctn\} \land ctn.type = \text{"input"})\)
   
i.e. All input nodes in the data flow graph are scheduled on the input node in the control graph.

2. \((\forall \: nl \in nL \land nl.df.type = \text{"output"} : nl.ct = \{ctn\} \land ctn.type = \text{"output"})\)
   
i.e. All output nodes in the data flow graph are scheduled on the output node in the control graph.

3. \((\forall \: nl \in nL \land nl.df.type = \text{"output"} \land ctn \in nl.ct \land e \in dfg.Edges\land
   e.destnode = nl.df : I(nl.df, ctn) = 0)\)
   
i.e. The delay index for the output nodes must be 0. This implies that the outputs of the scheduled data flow graph must be ready when the control graph reaches its output node.

4. \((\forall \: nwn \in nwg.Nodes \land nwn.sequ \land nl_1, nl_2 \in nL \land nl_1.nw = nl_2.nw = nwn\land
   ctn \in nl_1.ct \land ctn \in nl_2.ct \land dfn_1 = nl_1.df \land dfn_2 = nl_2.df\land
   I(dfn_1, ctn) = nl_1.delay \land I(dfn_2, ctn) \neq \infty : dfn_1.cond \land dfn_2.cond = \text{false})\)
   
i.e. A sequential network device cannot be started for two different purposes, during the same state and condition.

5. \((\forall \: nwn \in nwg.Nodes \land nwn.comb \land nl_1, nl_2 \in nL \land nl_1.nw = nl_2.nw = nwn\land
   ctn \in nl_1.ct \land ctn \in nl_2.ct \land dfn_1 = nl_1.df \land dfn_2 = nl_2.df\land
   I(dfn_1, ctn) \neq \infty \land I(dfn_2, ctn) \neq \infty : dfn_1.cond \land dfn_2.cond = \text{false})\)
   
i.e. A combinatoric network device cannot be used for two different purposes, during the same state and condition.

Notice that the last two rules are a result of the fact that a network device cannot be used to implement two data flow nodes in the same control node, under same conditions. This constraint could be weakened a little bit by stating that a network device may be used by two data flow nodes at a moment under the same conditions if the data flow nodes represent the same operation and have the same inputs. By considering the functionality of the data flow nodes, one could even weaken these condition more. For example, if there is a path between one value node dfn_1 to another value node dfn_2. Then these value nodes may be mapped on the same register nwn in a state ctn, even if

\[ I(dfn_1, ctn) = nl_1.delay \land I(dfn_2, ctn) \neq \infty \land (dfn_1.cond \land dfn_2.cond \neq \text{false}) \]

But these are special cases and will not be considered, since our goal is to establish a more global set of rules for the use of nodelinks.
Chapter 7

Interconnect and controller generator

7.1 Introduction

In the previous chapters we have talked about the data flow, control and network graphs. We have seen how nodelinks are used to store the scheduling and binding information in a design. And we have also given the rules how to use these nodelinks, by using the delay index and data flow conditions. Thus, we know how to interpret the objects in the design we get as input for our interconnect and controller generator. Now our job is to build the interconnect and controller generator.

To make things simpler, we have divided the interconnect and controller process into a few subprocesses. The process is divided into the parts as shown in figure 7.1.

The first subprocess maps all paths in the data flow graph onto edges in the network graph. To each network edge we attach a network function, which stores the information about when the network edge must be used. This information will be generated and stored when a data flow path is mapped onto a network edge.

After the first process the network graph edges are representing paths of the data flow graph. Signal conflicts may occur on ports with two connected network edges. To solve this, multiplexers are added to the network graph by the second subprocess.

Now, the data flow part of the network is finished. All data flow paths are represented in the network graph and there are no signal conflicts in the network graph. The 3rd process will create the edges representing control connections to and from the controller. These connections are required to control the devices in the network. In the network functions of the created network edges, we store the information about the purpose of the connection.

The network functions of the network edges connected to the controller, contain the information about the behavior of the controller. Together with the control graph this gives us enough information to extract a description of the finite state machine representing the controller. This is done in the 4th subprocess.
CHAPTER 7. INTERCONNECT AND CONTROLLER GENERATOR

Scheduled and Bound Design

1. Map Data Flow Paths on Network Edges

2. Solve Signal Conflicts in Network Graph

3. Create Control Connections in the Network Graph

4. Extract Finite State Machine describing Controller

5. Extract Net and Module Lists from Network Graph

Figure 7.1: Scheme of our interconnect and controller generator
In the last process we extract the net and module lists from the network graph.

7.2 Mapping data flow paths onto network edges

The first phase in the interconnection process is to map each data path in the data flow graph on a network edge in the network graph. We will also determine the network conditions under which the edge is used for data transport. To determine under which states and conditions a network edge (connection) is used, we have to determine the states and conditions a data flow path is scheduled in. This is done by looking at the delay index of the destination data flow node of the path.

7.2.1 Network conditions

Just like data flow conditions in the data flow and control graphs, we need to have a way to represent conditions in the network graph. Given a scheduled and bound graphlink it is easy to determine the corresponding network condition for an arbitrary data flow condition. This is because each condition generating port (pin) in the data flow graph is mapped on a network port (pin) in the network graph.

**Definition 7.1** (Condition generating network port)

Let there be a graphlink \( gl = (dfg, nwg, ctg, nL) \) with \( gl.sched = true \land gl.bound = true \). Let \( port \) be a port of a node in \( nwg \). Then \( port \) is a network condition generating port if,

\[
\exists \ nL \in nL \land port.\text{owner} = nL.n wn \\
: nl.dfport(port) is a condition generating data flow port
\]

**Definition 7.2** (Condition generating ports in network graph)

Let there be a network graph \( nwg \). Then the set of all condition generating ports in this graph is denoted by \( nwg.condPorts \).

Each port in the network graph contains a set of pins. We will define the network condition as a boolean expression, in which each pin (bit) of a condition generating port corresponds to a literal (boolean variable) in the expression.

**Definition 7.3** (Literal function)

Let there be a network graph \( nwg \). Then the literal (boolean variable) function \( nwg.lit \) is a bijection from the pins of condition generating network ports in \( nwg \) onto a set of boolean variables \( B \).

\[
nwg.lit : (\cup : port \in nwg.condPorts : port.Pins) \rightarrow B
\]
Definition 7.4 (Network condition)
Let there be a network graph $nwg$ and let $B$ be the range of its literal function $nwg.lit$. Then a network condition is a boolean expression, in which the literals are elements of $B$.

Definition 7.5 (Condition space of network graph)
Let there be a bound and scheduled graphlink $gl = (dfg, nwg, ctg, nL)$. Then the condition space of the network graph $nwg$, denoted by $nwg.Conds$ is the set of all possible network conditions.

Definition 7.6 (Condition as set of productterms)
Let $c$ be a network condition. Then this condition can be expressed as a sum of products and $c.Terms$ is the set of those productterms.

7.2.2 Transformation of data flow conditions into network conditions
It should be clear that the transformation of a data flow condition into a network condition is a mapping of one data flow literal onto one network literal. Notice that two data flow literals could be mapped on one network literal (because several data flow nodes can be mapped onto one network node). However, when transforming the data flow conditions obtained by the method presented in section 6.2, it should never occur that two different literals in one data flow condition are mapped onto a same literal in the network condition. If this happens anyway, then we know that something has gone wrong in the previous HLS processes.

So, in transforming our data flow conditions into a network condition, we can simply replace each data flow literal with its network's counterpart.

7.2.3 Network function
In the data flow graph the condition for an edge to transport tokens is given by the data flow condition. In the network graph we need, beside the network condition, also to know the state in which a network edge (connection between two ports) is needed for signal transport. To indicate this, we will attach a network function to each network edge.

Definition 7.7 (Network function)
Let there be a bound and scheduled graphlink $gl = (dfg, nwg, ctg, nL)$. Then a network function $f$ is a set of clauses. Here a clause is a 2-tuple consisting of a network condition and a control state,

$$f \subseteq nwg.Conds \times ctg.Nodes$$

Definition 7.8 (Network function of network edge)
Let there be a network edge $nwe$. Then the network function of this edge is denoted by $nwe.func$. 
The network function of a network edge tells us when and under what conditions there must be a connection between the origin and destination port. An edge with network function \( f \) which is empty, means that the edge is not used at all and therefore will not result in any connections in the final network.

### 7.2.4 Algorithm

We will now present the algorithms to map data flow paths onto network edges. The data flow paths with a destination node linked to combinatoric devices, will be handled by algorithm 7.1. The data flow paths "to" sequential devices will be mapped on network edges using algorithm 7.2.

**Algorithm 7.1 (The mapping of combinatoric data flow paths onto network edges)**

```cpp
GraphLink::mapCombDfPathsOnNwEdges()
{
    /* Pre: this = (dfg,nwg,ctg,nL) */
    for all (nl \in nL \land nl.nw.comb \land p \in dfg.Paths \land p.type = "data" \land p.dest = nl.df \land
              ctn \in nl.ct \land ctn.type = "state"|"input"|"output" \land I(nl.df,ctn) \neq \infty \land
              (p.cond \land ctn.cond) \neq false)
    {
        nwe = nwg.getNwEdge(p,this)
        nwcond = transformToNetworkCondition(p.cond \land ctn.cond)
        nwe.func = nwe.func \cup \{(nwcond,ctn)\}
    }
end all
}
```

In algorithm 7.1, we iterate over all nodelinks linked to combinatoric network devices. We then examine all the data flow paths to the data flow nodes linked to these nodelinks. Of these paths only the "data" paths may result in a network connection. A path to the input port of a constant node for instance, is a source path and thus do not have to be mapped on a network edge. During which control nodes the connection should be established is determined by the delay index function \( I(nl.df,ctn) \neq \infty \). We will only consider the "state", "input" and "output" control nodes in evaluating the delay index. When the condition of a data flow path and the condition of the control node are conflicting \( ((p.cond \land ctn.cond) = false) \), it is not possible to have that path represented in the network, during that control node.

The function `getNwEdge` returns a network edge, given a data flow path and graphlink. The origin network port of this network edge will correspond to the origin data flow port of the data flow path according to the other nodelinks in the graphlink. `getNwEdge` will create such a network edge in the network graph, if it does not exist yet. If such a network edge already exists, then that edge will be returned. By using `getNwEdge` we are sure that there is always at most one network edge between two network ports. Notice that when a network edge is created, the attached network function is empty.
The network condition under which the network connection must be established is determined by transformation of the product of the data flow path condition and the control node condition. In the for all-condition we have already stated that the argument for transformToNetworkCondition unequals false. Thus, the resulting network condition is also not false.

The obtained network condition and control node forms a network clause. This clause is added to the set of clauses in the edge's network function.

Algorithm 7.2 (The mapping of sequential data flow paths onto network edges)

```cpp
GraphLink::mapSequDfPathsOnNwEdges()
{
    /* Pre: this = (dfg, nwg, ctg, nL)*/
    for all (nl ∈ nL ∧ nl.nw.sequ ∧ dfp ∈ dfg.Paths ∧ dfp.type = "data" ∧
        dfp.dest = nl.df ∧ ctn ∈ nl.ct ∧ ctn.type = "state" | "input" | "output" ∧
        I(nl.df, ctn) = nl.delay)
    {
        for all (ctp ∈ ctg.Paths ∧ (ctp.cond ∧ dfp.cond) ≠ false ∧ ctp.dest = ctn)
        {
            nwe = nwg.getNwEdge(dfp, this)
            nwcond = transformToNetworkCondition(dfp.cond ∧ ctp.cond)
            nwe.func = nwe.func ∪ {(nwcond, ctp.orig)}
        }
    }
}
```

Algorithm 7.2 is more complicated than the previous algorithm. Notice that the outer loop in algorithm 7.2 is almost the same as the loop in algorithm 7.1. The main difference is that we now consider sequential devices and the control nodes with \(I(df_n, ctn) = nl.delay\) instead of \(I(df_n, ctn) ≠ \infty\). Now, in the innerloop \(ctn\) is the control node during which the sequential device begins its execution. The connections to this sequential device must be established in the preceding control nodes ("state" or "input" node). Thus, we iterate in the innerloop through the control paths leading to \(ctn\). Then, if the control paths condition is compatible with the data flow paths condition (\((ctp.cond ∧ dfp.cond) ≠ false\)), we will make a network connection in a similar way as in algorithm 7.1.
7.3 Solve signal conflicts

In this phase of the interconnection process, multiplexers are added to the network graph in order to solve signal conflicts in the network graph. For each network port that is the destination of more than one network edge, we will use a multiplexer to resolve that. For inout ports in the network that are used as input as well as output ports, we have to do something extra. When the inout port is configured as output port, then it can be connected with another output port. This is solved by using tri-state buffers.

7.3.1 Multiplexer

A multiplexer is a network device with several data inputs, a control input and one data output. Depending on the signal at the control port, one of the data input ports is connected to the output port.

The data input ports of a multiplexer are named 0, 1, 2, .... The signal on the control port "control" specifies the binary value of the input port to be connected to the output. If the signal on the control port is the binary value of a not existing input port, then an arbitrary data input port is connected to the output. For instance, if mux3 is a multiplexer with three data inputs, then these are named 0, 1 and 2. If the bitvector 01 appears at the control port, then port 1 will be connected to the output. But if the signal had been 11, then any of the ports 0, 1 or 2 may be connected to the output.

The type of the multiplexers indicates the number of data input ports, because the type is formed by concatenating "mux" with the number of data input ports. Thus, "mux8" is a multiplexer with 8 data input ports.

Definition 7.9 (Operators on network node related to multiplexers)

Let \textit{nwn} be a network node. To indicate whether a network node is a multiplexer, the operator \textit{.mux} is defined as,

\[
\textit{nwn.mux} = \begin{cases} 
\text{true} & \text{if nwn is a multiplexer} \\
\text{false} & \text{otherwise}
\end{cases}
\]

Definition 7.10 (Control bitvector for multiplexer)

Let \textit{nwn} be a network node and \textit{nwn.mux} = \textit{true}. Then the operator \textit{.bitvec} is defined as,

\[
\textit{nwn.bitvec : nwn.inPorts\{nwn.port("control")\} \rightarrow Bitvectors}
\]

\[
\textit{nwn.bitvec(port)} = \text{the bitvector of the signals that must appear on nwn.port("control") to have port connected to the output}
\]

Algorithm 7.3 shows how multiplexers are added to the network. In the outerloop a networkport, that is the destination of more than one network edge is picked. Depending on the number of edges a suitable multiplexer is created by \textit{createMultiplexer}. Then a network
connection is made between the output port of the multiplexer, \textit{muxout} and \textit{port}. In the second loop the edges connected to \textit{port} are reconnected one by one to a data input port of the multiplexer. The network functions of the input edges of the multiplexer are copied into the network function of the output edge. This is needed to make the network functions of all network edges in the network graph consistent.

\textbf{Algorithm 7.3 (Add multiplexers to the network)}

\begin{verbatim}
IvGraph::addMuxes()
{
  /* Pre: this = nwg */
  for all (nwn ∈ nwg.Nodes ∧ port ∈ nwn.Ports ∧ port.type ≠ "output" ∧ #port.inEdges > 1)
  {
    inedges = port.inEdges
    mux = createMultiplexer(#inedges)
    muxout ∈ mux.outports
    outnwe = getNwEdge(muxout, port)
    for all (innwe ∈ inedges ∧ muxin ∈ mux.inPorts\{mux.port("control")\} ∧
      innwe.dest ≠ mux ∧ #muxin.Edges = 0)
    {
      innwe.destport = muxin
      outnwe.func = outnwe.func ∪ innwe.func
    }
  }
}
\end{verbatim}

\subsection{7.3.2 Tri-state}

A tri-state is a network device with one control input "control", one data input "in" and one data output port "out". The control port has a width of 1. When the signal high is set on the control port, then the input port is connected to the output port. When the signal on the control port is low, then the input port and output port are disconnected, and the output port is high impedant.

After adding the multiplexers, it could happen that there are inout ports in the network connected to an output or inout port of some other device. For each inout port that is the destination port of a network edge, we have to use a tri-state, to protect the inout port, in case it is configured as an output port.

Algorithm 7.4 is used to add the tri-states in the network graph.

In figure 7.2a we see a part of a network graph without multiplexers and tristates. Then using algorithm 7.3 the conflict at the input port of module 5 is solved but the inout ports
Algorithm 7.4 (Add tri-states to the network)

```cpp
NwGraph::addTriStates()
{
    /* Pre: this = nwg */
    for all (nwn ∈ nwg.Nodes ∧ port ∈ nwn.Ports ∧ port.type = "inout" ∧ nwe ∈ port.inEdges)
    {
        tri = createTriState()
        newedge = getEdge(tri.port("out"), port)
        nwe.destport = tri.port("in")
        newedge.func = nwe.func
    }
}
```

of module 4 and 5 still need some tristates (figure 7.2b). In figure 7.2c, we see the result of applying algorithm 7.4.
Figure 7.2: a. Situation before adding multiplexers. b. After adding multiplexers. c. After adding tristates.
CHAPTER 7. INTERCONNECT AND CONTROLLER GENERATOR

7.4 Interconnection for control purposes

In this phase we will connect all control ports in the network graph that are used for control purposes to the controller in the network. We will not only create the network edges, we will also store information in each edge about when the connections are needed and for what signals the connections are used.

7.4.1 Extension of network function

The clauses of the network function as defined in the definition 7.8 contain a network condition and a control node. For specifying the active conditions of connections between normal network devices, this is suitable. But for specifying connections to the controller we also need a field to indicate what kind of control signal is meant. We will therefore extend the network clause by adding a signal to it. The signal may have the values as presented on page 41. The already existing clauses, obtained in the previous processes, will get the signal "NoValue".

Definition 7.11 (Signal)
A signal is an element of Signals, with
\[
\text{Signals} = \{ \text{"low"}, \text{"high"}, \text{"don't care"}, \text{"NoValue"}, \text{"start"}, \text{"ready"}, \text{"clock"} \} \cup \text{Bitvectors}
\]

Definition 7.12 (Redefinition of network function)
Let there be a bound and scheduled graphlink \( gl = (dfg, nwg, ctg, nL) \). Then a network function \( f \) is a set of clauses. And a clause is a 3-tuple consisting of a network condition, a control state and a signal,
\[
f \subset nwg.\text{Conds} \times ctg.\text{Nodes} \times \text{Signals}
\]

7.4.2 Algorithm

The creation of "control interconnections" is divided into 4 parts, control connections to combinatoric devices, control connections to sequential devices, control connections to multiplexers and the control connections to the tri-states. Algorithm 7.5 will call the four procedures that generate these control connections for us.

Three more pins will be added to the controller, clock, ready and start. The clock input pin is used for connection to the network's system clock. The ready output pin is used by the controller to signal that the controller is reaching the standby-state or is already in that state. The start input pin is used to start the controller.

Algorithm 7.6 is used to create the interconnections from the controller to the combinatoric devices. In the outerloop a suitable network device and a control node is picked. In the innerloop the ports of that network device will be examined one by one.

The function \text{getSignal} returns the signal that should be connected to the network port in the argument according to a nodelink. \text{getSignal} do this by looking up the defining graphlink.
Algorithm 7.5 (Connect the control ports of all devices to their own port in the controller, and store the information about when and what kind of signals are expected in the corresponding network function.)

```cpp
/* Pre: this = (dfg, nwg, ctg, nL) */
ctrlToCombDevice()
ctrlToSeqDevice()
ctrlToMuxDevice()
ctrlToTriState()
nwg.addInputNode("start")
nwg.connectCtrlStart()
nwg.addOutputNode("ready")
nwg.connectCtrlReady()
nwg.addInputNode("clock")
nwg.connectCtrlClock()
}
```

Algorithm 7.6 (Make the connections between the combinatoric control ports and the controller)

```cpp
/* Pre: this = (dfg, nwg, ctg, nL) */
for all (nl ∈ nL ∧ nl.nw.comb ∧ nl.nw.mux = false ∧ ctn ∈ nl.ct ∧
  ctn.type = "state" | "input" | "output" ∧ \( I(df, ctn) \neq \infty \))
  { for all (port ∈ nl.nw.ports)
    { sig = nl.getSignal(port)
      if (sig = "NoValue") continue
      nwe = nwg.connectToCtrl(port)
      nwcond = transformToNetworkCondition(nl.df.cond ∧ ctn.cond)
      nwe.func = nwe.func \cup \{(nwcond, ctn, sig)\}
    }
  }
}
```
for its nodelink in the database. If the network port in the argument is not used for controller purposes, then getSignal will return "NoValue".

If sig \neq "NoValue", we know that the port port must be connected to the controller. The function connectToCtrl checks whether port is connected to the controller. If there already exists an edge between port and the controller, this edge will be returned. Otherwise connectToCtrl will create a new network port in the controller and a new network edge between that port and port. By using connectToCtrl we can be sure that there is always at most one network edge between a network port and the controller.

The condition under which the data flow node in the nodelink nl is executed in ctn is given by nl.df.cond \land ctn.cond. The network equivalent nwcond of this condition is obtained by transformToNetworkCondition. Then the clause (nwcond, ctn, sig) is added to the network function of the network edge.

To create the interconnections from the controller to the sequential devices, algorithm 7.7 is used. In this algorithm the outerloop serves the same purpose as in algorithm 7.6. ctn is the state in which the device starts its execution. In the second loop we are looking for a suitable preceding control node of ctn to setup the connections, such that at the transition from p.orig to ctn the control signals may be clocked into the device. p must meet the requirement (nl.df.cond \land p.cond) \neq false, because it is no use to set up the connections in p.orig while we know that the execution of nl.df can never occur simultaneously with the transition specified by p. The most innerloop is similar to the innerloop of algorithm 7.6.

In these loops the required start signals are handled too. But we have not specified when we do not wish a start-signal from the controller to a device. The procedure (noStartToSequDevice) will do this for us. Algorithm 7.8 will expand the function of the ”start-edges” to the controller, so that the signal ”low” is set on the start control ports of the devices. We want this to happen whenever \( I(nl.df, ctn) < nl.delay \), because then the network node bound to nl.df must not be disturb. Notice that the loops in algorithm 7.8 have the same structure as those in algorithm 7.7.

In algorithm 7.7 we have not considered the ready signals from the sequential devices yet. The procedure readyFromSequDevice will connect the ”ready-ports” of all sequential devices in the network to the controller. In the function of these edges we will record when the controller is supposed to receive a ready signal. Remember that we only have to connect the ready ports of modules with an integer as delay (see section 4.3.5). Thus, the ready signal need to be checked by the controller only if \( I(nl.df, ctn) = ctn.delay = 1 \).
Algorithm 7.7 (Make the connections between the sequential control ports and the controller)

GraphLink::ctrlToSequDevice()
{
    /* Pre: this = (dfg, nwg, ctg, nL) */
    for all (nl ∈ nL ∧ nl.nw.sequ ∧ ctn ∈ nl.ctn ∧
        ctn.type = "state" | "input" | "output" ∧ I(nl.df, ctn) = nl.delay)
    {
        for all (p ∈ ctg.Paths ∧ (nl.df.cond ∧ p.cond) ≠ false ∧ p.dest = ctn)
        {
            for all (nwp ∈ nl.nw.ports)
            {
                sig = nl.getSignal(nwp)
                if (sig = "NoValue" | "ready" | "clock") continue
                nwe = nwg.connectToCtrl(nwp)
                nwcond = transformToNetworkCondition(nl.df.cond ∧ p.cond)
                nwe.func = nwe.func ∪ {(nwcond, p.orig, sig)}
            }
        }
    }

    noStartToSequDevice()
    readyFromSequDevice()
}
Algorithm 7.8 (Expand the functions of the "start signal" network edges, such that no start signal is generated when device is occupied.)

GraphLink::noStartToSequDevice()
{
    for all (nl \in nL \land nl.nw.sequ \land ctn \in nl.ct\n        \land ctn.type = "state" \land \neg \lnot "output" \land \neg \lnot I(nl.df, ctn) < nl.delay)
    {
        for all (p \in ctg.Paths \land (nl.df.cond \land p.cond) \neq \false \land p.dest = ctn)
        {
            for all (nwp \in nl.nw.ports)
            {
                sig = nl.getSignal(nwp)
                if (sig \neq "start") continue
                nwe = nwg.connectToCtrl(nwp)
                nwcond = transformToNetworkCondition(nl.df.cond \land p.cond)
                nwe.func = nwe.func \cup \{(nwcond, p.orig, low)\}
            }
        }
    }
}

Algorithm 7.9 (Connect the ready ports of sequential devices to the controller.)

GraphLink::readyFromSequDevice()
{
    for all (nl \in nL \land nl.nw.sequ \land ctn \in nl.ct\n        \land ctn.type = "state" \land \neg (nl.df, ctn) = 1)
    {
        for all (nwp \in nl.nw.ports)
        {
            sig = nl.getSignal(nwp)
            if (sig \neq "ready") continue
            nwe = nwg.connectToCtrl(nwp)
            nwcond = transformToNetworkCondition(nl.df.cond \land p.cond)
            nwe.func = nwe.func \cup \{(nwcond, p.orig, sig)\}
        }
    }
}
We have discussed the control signals to the sequential and combinatoric devices so far. When we discussed the combinatoric devices we have skipped the multiplexers and tri-states, because they need a special treatment. Algorithm 7.10 is used for the creation of the connection to control the operation of the multiplexers in the network. Which input port of the multiplexer is switched to the output depends on the signal set on the control port of the multiplexer. The bitvector representation of this signal corresponds to the binary value of the decimal named ports. The function `getMuxSignal()` will return the signal corresponding to `port`.

Algorithm 7.10 (Make the connections between the multiplexer control ports and the controller)

GraphLink::ctrlToMuxDevice()
{
    /* Pre: this = (dfg,nwg,ctg,nL) */
    for all (nwn ∈ nwg.Nodes ∧ nwn.mux = true)
    {
        nwe = nwg.connectToCtrl(nwn.port("control"))
        for all (port ∈ nwn.inPorts ∧ port ≠ nwn.port("control")∧
            (cond,ctn,s) ∈ port.edge.func)
            {
            sig = bitvector(port.name)
            nwe.func = nwe.func ∪ (cond,ctn,sig)
            }
        end all
    }
end all
}

The connections to the control ports of the tri-states are created by algorithm 7.11. In the created edge we record all the condition and states under which the tri-state should be in conducting mode (signal high on control port).
Algorithm 7.11 (Make the connections between the tri-states control ports and the controller)

```
GraphLink::ctrlToTriState()
{
    /* Pre: this = (dfg, nwg, ctg, nL) */
    for all (nwn ∈ nwg.Nodes ∧ nwn.type = "tri-state")
    {
        nwe = nwg.connectToCtrl(nwn.port("control"))
        for all ((cond, ctn, s) ∈ nwn.port("in").edge.func)
        {
            nwe.func = nwe.func ∪ (cond, ctn, high)
        }
        end all
    } end all
}
```
7.5 Extract finite state machine describing the controller

In this stage of the interconnection process we will extract a finite state machine from the control graph and the information stored in the network functions of the edges connected to the controller. The finite state machine will be specified in the form of a TAB [Huijnen91] table (section 7.5.1). In our algorithms we will use bitvectors to represent the rows of a TAB table. We will therefore pay some attention on the bitvectors in section 7.5.2, before we discuss the algorithms in following sections.

7.5.1 TAB table

The behavior of the controller needs to be specified in the form of a TAB table. In a TAB table each row specifies what the outputvector and next state should be given a current state and inputvector of the controller. Each element in the inputvector and outputvector corresponds to one input respectively output pin of the controller. And each pin of the controller is represented by exactly one element in the input or outputvector. Each element in the vector may have the value 0, 1 or '−'. 0 stands for the signal "low", 1 for "high" and '−' for "dont-care".

Definition 7.13 (TAB table)
A TAB table is a set of rows, that specifies a finite state machine.

Definition 7.14 (Row of TAB table)
Let \( \text{ctrl} \) be a network node representing a controller. Then a row of a TAB table specifying that controller, is a 4 tuple consisting of a current state, input-bitvector, output-bitvector and next state. Let there be a row \( \text{row} = (\text{cs}, \text{inv}, \text{outv}, \text{ns}) \), with \( \text{cs} \) as the currents state, \( \text{inv} \) as the input vector, \( \text{outv} \) as the outputvector and \( \text{ns} \) the next state. Then the operators \( .cs \), \( .in \), \( .out \) and \( .ns \) are defined on this row as

- \( \text{row}.cs = \text{cs} \)
- \( \text{row}.in = \text{inv} \)
- \( \text{row}.out = \text{outv} \)
- \( \text{row}.ns = \text{ns} \)

The number of elements of the input bitvector corresponds of course to the number of input pins of \( \text{ctrl} \). This is also true for the output bitvector and the output pins of \( \text{ctrl} \).

Definition 7.15 (Complete specified bitvector)
A complete specified bitvector is a bitvector in which all entries are '0' or '1'. Thus given a vector \( v \) and \( \#v = n \), there exist \( 2^n - 1 \) other complete specified bitvectors with the same number of elements\(^1\). The set of all complete specified bitvectors with \( n \) elements is denoted as

\[ \text{Bit}^n \]

\(^1\)\#v denotes the number of entries in bitvector \( v \)
Definition 7.16 (Containment of bitvectors)
Let there be two bitvectors $v_1$ and $v_2$, with $\#v_1 = \#v_2$. Then $v_1$ is contained in $v_2$ if and only if
\[
(\forall 0 \leq i < \#v_1 : v_2[i] = ' -' \lor v_2[i] = v_1[i])
\]
If $v_1$ is contained in $v_2$, one could also say that $v_1$ implies $v_2$, also denoted as
\[
v_1 \Rightarrow v_2
\]

Definition 7.17 (Overlapping rows)
Let there be two rows in a TAB table, $row_1 = (cs_1, inv_1, out_1, ns_1)$ and $row_2 = (cs_2, inv_2, out_2, ns_2)$. Then $row_1$ and $row_2$ are overlapping if and only if
\[
(\forall 0 \leq i < \#inv_1 : inv_1[i] = inv_2[i] \lor inv_1[i] = ' -' \lor inv_2[i] = ' -') \land cs_1 = cs_2
\]

There is one important restriction on a generated TAB table. Let $table$ be a TAB table in which the input bitvectors have $n$ elements. For each state $s$ of the controller specified by $table$, it is required that,
\[
(\forall bitv \in Bit^n : (\exists row \in table \land row.cs = s : bitv \Rightarrow row.in))
\]
This means that the space spanned by the inputs and states of the controller must be covered completely by the table. And there may be no overlapping rows in the table ($\exists!$).

Notice that one finite state machine can be specified by many different TAB tables. If a controller has $n$ input pins and $S$ states, then the TAB table will contain at most $2^n \cdot S$ rows. However, the goal is of course to generate a table with as less rows as possible.

7.5.2 Bitvector calculus

In this section we will show some operations on bitvectors. This is important because in the algorithms we will often manipulate with bitvectors stored in the spec's of a control node. One should bear in mind the resemblance between a productterm of a boolean expression and a bitvector. A boolean expression can always be rewritten as a sum of products. Each such productterm may be specified by a bitvector, in which each element corresponds to a literal in the expression. An entry '0' means that the negation of the corresponding literal is present in the productterm. An entry '1' means that the corresponding literal itself is present in the productterm. An entry '−' means that neither the negation as the literal itself is present in the productterm. In this way, a boolean expression can be described by a set of bitvectors.

In this section we will define some often used functions for operations on bitvectors. The most important one is the modification of the specs (see section 7.5.3) in control nodes by splitting a bitvector into a set of other bitvectors that forms a partitioning of the space covered by the first bitvector. We will first define other handy functions, and use these in the definition of the split function.
Definition 7.18 (Right)
Let there be a bitvector $v$ and $\# v = n$ and a non-negative integer $i$ then the function $\text{right}(v, i)$ is defined as,

$$\text{right} : \text{Bit}^n \times \mathbb{N} \rightarrow \text{Bit}^n$$

$$\text{right}(v, i) = \begin{cases} v & \text{if } n > i \land v[i] = \text{'-'}, \\ v_1 & \text{if } n \leq i, \\ v_2 & \text{otherwise} \end{cases}$$

with $v_1$ and $v_2$ meeting the requirements:

- $(\forall : 0 \leq j < n : v_1[j] = \text{'-'}$)
- $(\forall : 0 \leq j < i : v_2[i] = v[i])$
- $(\forall : i < j < n : v_2[i] = \text{'-'})$
- $(v_2[i] = \text{'1'} \land v[i] = \text{'0'}) \lor (v_2[i] = \text{'0'} \land v[i] = \text{'1'})$

What the function $\text{right}$ does is setting all the entries of the elements with index larger then $i$ at `'-'` and inverse the entry of the element with index $i$.

Definition 7.19 (Inverse)
Let there be a bitvector $v$ and $\# v = n$, then the function $\text{inverse}(v)$ is defined as,

$$\text{inverse} : \text{Bit}^n \rightarrow 2^{\text{Bit}^n}$$

$$\text{inverse}(v) = (\cup : 0 \leq i < n \land v[i] \neq '\text{'-'}' : \{\text{right}(v)\})$$

$\text{inverse}$ may also be denoted by overlining a bitvector, $\text{inverse}(v) = \overline{v}$.

The result of $\text{inverse}(v)$ is a set of bitvectors, which forms a partitioning of the boolean space not covered by $v$. The cardinality of $\text{inverse}(v)$ is the same as the number of entries inequal `'-'` in $v$. One may verify easily, that the inversion performed by the $\text{inverse}$ function is based on the following rule in boolean algebra:

$$\overline{x_0x_1x_2\ldots x_n} = \overline{x_0} + x_0\overline{x_1} + x_0x_1\overline{x_2} + \ldots + x_0x_1x_2\ldots\overline{x_n}$$

Definition 7.20 (Intersect)
Let there be two bitvectors $v_1$ and $v_2$, with $\# v_1 = \# v_2 = n$. Then the function $\text{intersect}(v_1, v_2)$ is defined as,

$$\text{intersect} : \text{Bit}^n \times \text{Bit}^n \rightarrow 2^{\text{Bit}^n}$$

$$\text{intersect}(v_1, v_2) = \begin{cases} \emptyset & \text{if } (\exists : 0 \leq i < n : (v_1[i] = \text{'0'} \land v_2[i] = \text{'1'}) \lor (v_1[i] = \text{'1'} \land v_2[i] = \text{'0'})) \\ \{v_3\} & \text{otherwise} \end{cases}$$

with $v_3$ such that

$$(\forall : 0 \leq i < n : (v_1[i] = \text{'-'} \land v_3[i] = v_2[i]) \lor (v_2[i] = \text{'-'} \land v_3[i] = v_1[i]) \lor (v_1[i] = v_2[i] = v_3[i]))$$

$\diamondsuit$
The function \textit{intersect} gives us if possible a set containing a bitvector that implies the two bitvectors in its argument. If the two bitvectors in the argument are conflicting, then an empty set is returned.

\textbf{Definition 7.21 (Split)}

Let there be two bitvectors $v_1$ and $v_2$, with $\#v_1 = \#v_2 = n$. Then the function \textit{split} is defined as,

\[
split : \text{Bit}^n \times \text{Bit}^n \rightarrow 2^{\text{Bit}^n}
\]

\[
split(v_1, v_2) = \text{intersect}(v_1, v_2) \cup (\cup : v \in \overline{v_2} : \text{intersect}(v_1, v))
\]

\textdiamond

In other words, the function \textit{split} will partition the boolean space covered by $v_1$ such that the overlapping and the nonoverlapping space with $v_2$ can be described as a subset of the resulting set of bitvectors.

\subsection*{7.5.3 Spec's of control node}

The controller graph will be transformed into a finite state machine with the following correspondence between the control nodes in the graph and states in the machine:

1. Each state node of the control graph corresponds to a state in the finite state machine.
2. The output node will become the standbystate in the finite state machine.

We will attach to each control node, that will become a state in the finite state machine, a set of so called spec's. We will then process the information generated by the previous interconnection steps, which is stored in the functions attached to the network edges connected to the network controller. The spec's will be modified during this process, so that eventually, the spec's in the control nodes can be mapped easily onto rows in a TAB table.

\textbf{Definition 7.22 (Spec)}

A spec is a 5-tuple, consisting of four bitvectors: input bitvector, ready bitvector, start bitvector, output bitvector, and a control node. The control node to which a spec \textit{spec} is attached will be denoted by \textit{spec.owner}. The input, ready, output and start bitvectors in \textit{spec} are denoted by respectively \textit{spec.in}, \textit{spec.ready}, \textit{spec.out} and \textit{spec.start}.

\textdiamond

\textbf{Definition 7.23 (Spec's of control node)}

Let \textit{ctn} be control node. Then the set of specs attached to \textit{ctn} is denoted by \textit{ctn.Specs}.

\textdiamond

Notice the resemblance between a spec and a row in the TAB table. The idea is that each spec will eventually become a row in the TAB table. A spec \textit{spec} is then mapped onto a row \textit{row} based on the following,

- $\textit{row.cs} = \textit{spec.owner}$
CHAPTER 7. INTERCONNECT AND CONTROLLER GENERATOR

- row.ns = spec.ctn
- row.in = spec.in : spec.ready
- row.out = spec.out : spec.start

Notice that the bitvector row.in is divided into two bitvectors spec.in and spec.ready. Each pin represented by an element of spec.ready, is a pin connected to the "ready port" of a sequential device. All other input pins of the controller are represented in spec.in. In a similar way, row.out is divided into a bitvector representing the pins connected to the "start ports" of the devices spec.start and a bitvector representing the other output pins spec.out. The reason for dividing these bitvectors will become clear in the sequel.

Algorithm 7.12 shows the procedure createTABLE, which will create the TAB table for us. First, all control nodes that will become a state in the finite state machine are initialized with one spec (algorithm 7.13). The functions makeInputVector, makeReadyVector, makeOutputVector and makeStartVector will return a bitvector with the proper length. For example, if the controller has 10 input pins and 3 of them are used to receive "ready" signals, then makeInputVector will return a bitvector of 7 elements and makeReadyVector will return a bitvector of 3 elements. When the argument true is passed to a make...Vector function, the entries of the returned bitvector will all be ' -'. The control node errorstate is created in the control graph to cover the unspecified transitions (page 28). Thus, the control nodes are initialized with all don't-care vectors and the errorstate as next state.

Algorithm 7.12 (Create TAB table)

GraphLink::createTABLE()
/* Pre: this = (dfg, nwg, ctn, nL) */
{
  ctn.initializeSpecs()
  ctn.splitOnNextStates()
  ctn.splitOnCondition()
  ctn.collectReadySignals()
  ctn.completeReadySpace()
  ctn.generateReady()
  ctn.disableTrisStates()
  ctn.exportTABfile()
}

Before we continue with the explanation of the procedures called by createTABLE, we will discuss a very often called procedure for modifying the set of specs in a control node.

Modification of Specs

The most often used procedure in creating a TAB table is modifySpecs. modifySpecs will modify the set of specs of its object (ctn), according to its arguments. There is one very
Algorithm 7.13 (Initialize specs)

CtGraph::initializeSpecs()
{
/* Pre: this = ctg */
invec = makeInputVector(true)
readyvec = makeReadyVector(true)
outvec = makeOutputVector(true)
startvec = makeStartVector(true)
errorstate = newCtNode("state")
for all (ctn ∈ ctg.Nodes ∧ ctn.type = "state"|"output")
{
  ctn.Specs = {(invec, readyvec, outvec, startvec, errorstate)}
}
}

important restriction on the result after modification, which we will discuss after giving the
definition of the set of spec's as partitioning of the input space.

Definition 7.24 (Set of specs as a partitioning of input space)
Let there be a controller, with \( n \) input pins and \( r \) of them are connected to "ready-ports",
the input bit vectors of the specs will then contain \( n - r \) elements. Then a set of spec's of a
control node \( ctn \) is said to form a partitioning of the input space (excluding the ready input
space) if and only if,

\[
(\forall: \text{bitvec} \in \text{Bit}^{n-r} : (\exists!: \text{spec} \in ctn.Specs : \text{bitvec} \Rightarrow \text{spec.in}))
\]

The modification of the specs of a control node will be done by modifySpecs in such a way
that the set of specs will always form a partitioning of the input space (excluding the ready
input space) if the set of specs already was a partitioning of the input space before calling
modifySpecs. This last condition is met, because the set of specs have been initialized as a
set of one spec covering the complete input space, and all modifications to the set of specs is
done by modifySpecs.

There are two versions of this procedure, we will first discuss the one presented in algo­
rithm 7.14. The arguments consist of four bitvectors, \( \text{invec, readyvec, outvec, startvec} \) and
a control node \( \text{ns} \). The goal is to modify the set of specs for a control node \( ctn \) such that,

1. \( (\forall: \text{spec} \in ctn.Specs \land \text{spec.in} \Rightarrow \text{invec} \\
    : \text{spec.out} \Rightarrow \text{outvec} \land \text{spec.start} \Rightarrow \text{startvec} \land \text{spec.ct} = \text{ns}) \)

2. \( (\forall: \text{spec}_1, \text{spec}_2 \in ctn.Specs \land \text{spec}_1 \neq \text{spec}_2 \\
    : \text{intersect(\text{spec}_1.\text{cond} \cdot \text{spec}_1.\text{ready}, \text{spec}_2.\text{cond} \cdot \text{spec}_2.\text{ready}) = \emptyset}) \)

and of course the partitioning in the input space must be maintained. The first requirement,
tells us that the bitvectors \( \text{outvec, startvec} \) and next state \( \text{ns} \) should be implied for all specs,
with an inputvector that implies `invec`. The second requirement states that their should be no overlapping in the complete input space (including the ready input space) of the specs.

At the first line in algorithm 7.14, we make an empty set of specs, which will be filled during the iterations, and assigned to the set of specs of the control node (`ctn`) at the last line. There are three nested loops in the body. In the most outer loop we take a spec in the existing set of specs of `ctn`. In the second loop we split that spec by `invec`. This results in a partitioning of the space covered by `spec.in`. For each bitvector in this partitioning we look whether it is overlapping with `invec`. This is done by examining `invecs`. If `invecs` is empty, we know `split_invec` is not overlapping with `invec` and therefore a spec with `split_invec` as inputvector is formed with the other fields the same as `spec`, and added to the set of `newspecs`. If `invecs ≠ ∅`, then we know that `invecs` contains exactly one bitvector, which is contained in `split_vector` as well as in `invec`. In this case we have to determine the other fields. `intersect(spec.ready, readyvec)` will give us a new ready bitvector if `spec.ready` and `readyvec` are overlapping. otherwise we take `readyvec` as the new ready bitvector. The new output and start vectors are then formed by `intersection of outvec and startvec` with the existing output and start vectors in `spec`. Now, if our design was scheduled and bound properly, it will never happen that `outvecs` or `startvecs` are empty. Then in the most innerloop (which is not really a loop because all the sets `invecs`, `readyvecs`, `outvecs` and `startvecs` only contain one element) a new spec is added to `newspecs`.

The second version of `modifySpecs` is given in algorithm 7.15. The only difference with the first version is that no control node is given as argument. Thus, the existing control node in each spec will be maintained in the specs that result from the split.
Algorithm 7.14 (Modify the set of spec’s in the control node. The set of spec’s will always form a partitioning of the input boolean space (the ready space excluded).)

\texttt{CtNode::modifySpecs(invec,readyvec,outvec,startvec,ns)}
{
    /* Pre: this = ctn */
    newspecs = 0
    for all (spec \in ctn.Specs)
    {
        for all (split.invec \in split(spec.in,invec))
        {
            invecs = intersect(split.invec,invec)
            if (invecs = 0)
            {
                newspecs = newspecsU
                {((split.invec,spec.ready,spec.out,spec.start,spec.ct})
                continue
            }
            readyvecs = intersect(spec.ready,readyvec)
            if (readyvecs = 0) readyvecs = \{readyvec\}
            outvecs = intersect(spec.out,outvec)
            startvecs = intersect(spec.start,startvec)
            if (outvecs = 0 V startvecs = 0)
            {
                error("conflicting outputs for controller")
            }
            for all (newinvec \in invecs \land newreadyvec \in readyvecs\land
            newoutvec \in outvecs \land newstartvec \in startvecs)
            {
                newspecs = newspecsU
                {((newinvec,newreadyvec,newoutvec,newstartvec,ns})
            }
        }
    }
    ctn.Specs = newspecs
}
Algorithm 7.15 (Modify the set of spec's in the control node. The set of spec's will always form a partitioning of the input boolean space (the ready space excluded).)

CtnNode::modifySpecs(invec, readyvec, outvec, startvec)
{
    /* Pre: this = ctn */
    newspecs = ∅
    for all (spec ∈ ctn.Specs)
    {
        for all (split_invec ∈ split(spec.in, invec))
        {
            invecs = intersect(split_invec, invec)
            if (invecs = ∅)
            {
                newspecs = newspecs ∪
                {
                    (split_invec, spec.ready, spec.out, spec.start, spec.ct)
                }
                continue
            }
            readyvecs = intersect(spec.ready, readyvec)
            if (readyvecs = ∅) readyvecs = {readyvec}
            outvecs = intersect(spec.out, outvec)
            startvecs = intersect(spec.start, startvec)
            if (outvecs = ∅ ∨ startvecs = ∅)
            {
                error("conflicting outputs for controller")
            }
            for all (newinvec ∈ invecs ∧ newreadyvec ∈ readyvecs ∧
            newoutvec ∈ outvecs ∧ newstartvec ∈ startvecs)
            {
                newspecs = newspecs ∪
                {
                    (newinvec, newreadyvec, newoutvec, newstartvec, spec.ct)
                }
            }
        }
    }
    end all
}
end all
ctn.Specs = newspecs
7.5.4 Expansion of literal function with start condition

Remember the network literal function of definition 7.3. This literal function mapped each pin of a condition generating network port onto a literal used to express the network conditions. Now, in the previous processes we have connected each condition generating network port to a port of the controller. Thus, we may say that each condition generating pin is mapped on one pin of an input port of the controller. Therefore, each pin of the input ports in the controller, not connected to "ready" ports and the start input node, corresponds to one literal in the network conditions. In other words, each entry in the input vector of a spec corresponds to one literal in network conditions, except the entry for the start input port (pin) in the controller. The start input port in the controller is connected to the network start input node. To make it easier for us to talk about this port among the other input ports, we will expand the literal function, such that the literal \( \sigma \) corresponds to the pin in the controller connected to the start node in the network.

**Definition 7.25 (Redefinition of literal function)**

Let there be a network graph \( nwg \) containing the controller node \( ctrl \). Then the literal (boolean variable) function \( nwg.lit \) is a bijection from the pins of condition generating network ports in \( nwg \) and the start pin in \( ctrl \) onto act set of boolean variables \( \{CT\} \).

\[
nwg.lit : (\cup : \text{port} \in nwg\text{.condPorts} : \text{port}\text{. Pins}) \cup \{ctrl\text{.port("start")}.0\} \rightarrow \{CT\}
\]

and

\[
nwg.lit(ctrl\text{.port("start")}.0) = \sigma
\]

Thus, \( \sigma \) corresponds to the condition that a start signal is given to the controller. And \( \overline{\sigma} \) is the condition that the signal low is on the input start pin of the controller. Notice that \( \sigma \) is in fact the same as the boolean variable \( \text{start} \) in section 3.3.

7.5.5 Specifying next states

The first procedure that is called by createTable after initialization is splitOnNextStates. This procedure is presented in algorithm 7.16. splitOnNextStates will modify the specs, such that the right control node, corresponding to the next state, is stored in the specs.

The \textit{ready}, \textit{output} and \textit{start} parts of the spec's are not important in this procedure. Thus, three bitvectors, \textit{readyvec}, \textit{outvec} and \textit{startvec} are created with the argument \textit{true}. In the first \texttt{for all} loop we specify that whenever the controller receives a start signal at a state transition, the next state will be one of the first states in the control graph. A first state in the control graph is a state that is the destination of a path in the control graph with the input node as origin. If there are more than one of such first state nodes in the control graph, then \( p\text{.cond} \) will tell us which one to choose. Because \( p\text{.cond} \) is a network condition, it could contain several productterms. Therefore we will iterate through these terms and make a bitvector \textit{invec} specifying that term. Because the controller must also receive a start signal, we use \( \textit{term} \land \sigma \) as argument to makeInputVector. \textit{invec} will be the bitvector representation of the productterm \( \textit{term} \land \sigma \). Then the function modifySpecs is called with all the bitvectors
and the destination of the path (next state) as arguments.

In the second for all loop we specify the next states, in case no start is given to the controller. Each path in the control graph not originating from the input node, will then represent one state transition.

In the third loop we specify that whenever the output node (= standby state) is reached, we will stay there until a start is given to the controller. After completion of splitOnNextStates, some specs in the control nodes could have the errorstate as next state. This will happen whenever there are no transitions specified for some nodes under certain conditions (not complete specified branches). Notice that the errorstate now has automatically itself as its next state if \( \overline{\sigma} \).

Algorithm 7.16 (Split the initial Spec's of all control nodes to cover the transitions to all possible next states.)

\[
\text{CtGraph}\text{.splitOnNextStates}()
\]

\{
/* Pre: this = ctg and nwg is the network graph in the design 
with the controller ctrl */
readyvec = makeReadyVector(true)
outvec = makeOutputVector(true)
startvec = makeStartVector(true)
for all (p ∈ ctg.Paths ∧ p.orig.type = "input" ∧
   term ∈ transformToNetworkCondition(p.cond).Terms ∧
   ctn ∈ ctg.Nodes ∧ ctn.type = "output"/"state")
   { 
invec = makeInputVector(term ∧ σ))
   ctn.modifySpecs(invec,readyvec,outvec,startvec,p.dest)
   }
end all
for all (p ∈ ctg.Paths ∧ p.orig.type ≠ "input" ∧
   term ∈ transformToNetworkCondition(p.cond).Terms)
   { 
invec = makeInputVector(term ∧ \overline{σ})
   p.orig.modifySpecs(invec,readyvec,outvec,startvec,p.dest)
   }
end all
for all (ctn ∈ ctg.Nodes ∧ ctn.type = "output")
   { 
invec = makeInputVector(\overline{σ})
   ctn.modifySpecs(invec,readyvec,outvec,startvec,ctn)
   }
end all
7.5.6 Specifying the output bitvectors

After splitOnNextStates, the procedure splitOnCondition is called by createTable. Algorithm 7.17 shows the contents of this procedure. splitOnCondition modifies the spec's such that the output vectors and start vectors are specified.

In the first loop, we look at all the control signals that have to be generated before the transition to a first state. Because a start signal to the controller is required for a transition to a first state, we restrict invec by $\sigma$. When the output signal ($clause.sig$) is a "start" signal from the controller to a device, then we have to specify $startvec$ and not $outvec$. makeStartVector($port$) returns a start vector with the entry of the element corresponding to $port$ set on '1' and all the other entries set on '-'. When the output signal ($clause.sig$) is not a "start" signal, then we need to specify $outvec$ and not $startvec$. makeOutputVector($port, sig$) returns an output vector in which the entries of the elements corresponding to $port$ have such values that the signal $sig$ is set on $port$.

In the second loop we will specify all the output and start bit vectors in the case that no start signal is given to the controller ($\bar{\sigma}$). The structure of this loop is similar to the first loop.
Algorithm 7.17 (Split the Spec's of all control nodes that is required to specify the outputs as a function of the current state and inputs.)

CtGraph::splitOnCondition()
{
    /* Pre: this = ctg and nwg is the network graph in the design
       with the controller ctrl */
    readyvec = makeReadyVector(true)
    for all (port ∈ ctrl.outPorts ∩ (cond, ct, sig) ∈ port.edge.func ∩
        term ∈ cond.Terms ∩ ct.type = "input" ∩
        ctn ∈ ctg.Nodes ∩ ctn.type = "output" ∩ "state")
    {
        invec = makeInputVector(term ∩ σ)
        if (sig = "start")
            {
            outvec = makeOutputVector(true)
            startvec = makeStartVector(port)
        }
        else
        {
            outvec = makeOutputVector(port, sig)
            startvec = makeStartVector(true)
        }
        ctn.modifySpecs(invec,readyvec,outvec,startvec)
    }
    end all
    for all (port ∈ ctrl.outPorts ∩ (cond, ct, sig) ∈ port.edge.func ∩
        term ∈ cond.Terms ∩ ct.type ≠ "input")
    {
        invec = makeInputVector(term ∩ σ)
        if (sig = "start")
            {
            outvec = makeOutputVector(true)
            startvec = makeStartVector(port)
        }
        else
        {
            outvec = makeOutputVector(port, sig)
            startvec = makeStartVector(true)
        }
        ct.modifySpecs(invec,readyvec,outvec,startvec)
    }
    end all
}
7.5.7 Specifying the ready bitvectors

After splitOnCondition, the procedure collectReadySignals and completeReadySpace is called by createTable. These two procedures are needed to specify the actions to be taken when the controller must verify the ready signals from some of the network devices.

The first procedure, collectReadySignals, is presented in algorithm 7.18. In the first loop we iterate through the input ports of the controller to look for a ready signal. In this loop we specify that the controller has to check the ready signal as given in readyvec under the condition given by invec. We have restricted invec with \( \overline{G} \), because when a start signal is given to the controller it is almost never required to check whether a device has finished its execution or not. makeReadyVector(port) returns a ready bitvector with the entry of the element corresponding to port set on 1 and all other entries set on '—'.

The second loop in collectReadySignals handles those cases we have neglected in the first loop by restricting term with \( \overline{G} \). There are in fact some cases in which the ready signals must be verified, even when a start signal is given to the controller. In all the last states of the control graph, we need to verify the ready signals even when a start signal is received by the controller (see also page 104). A control node is a last state when it is the origin node of a path with the output node as destination. During the last states of the control graph we have to check the incoming ready signals to decide whether the controllers ready signal should be generated or not. This is needed to indicate to the outside world whether the network is finishing its execution in the last state, even if the next state is a first state and not the standby state. Notice that the condition under which we need to check the ready signal is not only given by the condition under which the ready is generated (term), but also the condition whether it is going to be checked or not (transformToNetworkCondition(p.cond)).

In procedure collectReadySignals we have specified the when certain ready signals must be checked. The actions that should be taken when it turn out that a ready signal is not received by the controller at a moment that it should be generated is added to the specs by completeReadySpace. This routine will also make all the set of specs in the control nodes to be a partitioning not only in the space of input bitvectors, but also in the space of the ready bitvectors.

Algorithm 7.19 presents the contents of completeReadySpace. We iterate through all the control nodes that will be states in the finite state machine. Then we will look for specs in which some ready signals need to be verified, in which case \( \text{spec.ready} \neq \emptyset \). We will create a wait state for each control node that has such a spec. Then for each \( \text{readyvec} \) in the complementary space of \( \text{spec.ready} \) we modify the existing set with modifySpecs(spec.in, readyvec, spec.out, startvec, waitstate). startvec is spec.start in which all '1' entries are replaced with '0'-s. This is done because the start signal was meant to be setup for the execution in the next state. Since we are not going to make the transition to the next state yet, all the starts should be kept at '0'. When all the specs of the control node have been examined, we will copy its set of specs to its waitstate.
Algorithm 7.18 (Split the Spec’s of all control nodes, such that the ready signals can be collected.)

CtGraph::collectReadySignals()
{
    /* Pre: this = ctg and nwg is the network graph in the design
       with the controller ctrl */
    for all (port ∈ ctrl.inPorts ∧ (cond, ct, sig) ∈ port.edge.func
        term ∈ cond.Terms ∧ sig = "ready")
    {
      invec = makeInputVector(term ∧ σ)
      readyvec = makeReadyVector(port)
      outvec = makeOutputVector(true)
      startvec = makeStartVector(true)
      clause.ct.modifySpecs(invec, readyvec, outvec, startvec)
    }
    end all
    for all (port ∈ ctrl.inPorts ∧ (cond, ct, sig) ∈ port.edge.func
        term ∈ cond.Terms ∧ sig = "ready" ∧
        (∃ : p ∈ ctg.Paths : p.orig = ct ∧ p.dest.type = "output"))
    {
      tmpecond = (term ∧ transformToNetworkCondition(p.cond)).Terms
      for all (tmpterm ∈ tmpecond)
      {
        invec = makeInputVector(tmpterm ∧ σ)
        readyvec = makeReadyVector(port)
        outvec = makeOutputVector(true)
        startvec = makeStartVector(true)
        clause.ct.modifySpecs(invec, readyvec, outvec, startvec)
      }
      end all
    }
    end all
}
Algorithm 7.19 (Split the Spec's of all control nodes, such that the set of specs in each control node forms a partition of the input space including the ready space.)

\texttt{CtGraph::completeReadySpace()}
\{
  \texttt{/* Pre: this = ctg and nwg is the network graph in the design}
  \texttt{with the controller ctrl */}
  \texttt{ctns = ctg.Nodes}
  \texttt{for all (ctn \in ctns \land ctn.type = "state"|"output")}
  \{
    \texttt{waitstate = ctn}
    \texttt{specs = ctn.Specs}
    \texttt{for all (spec \in specs)}
    \{
      \texttt{if (spec.ready = \emptyset) continue}
      \texttt{if (waitstate = ctn) waitstate = newCtNode("state")}
      \texttt{for all (readyvec \in spec.ready)}
      \{
        \texttt{startvec = spec.start}
        \texttt{for all (0 \leq i < \#startvec \land startvec[i] = '1')}
        \{
          \texttt{startvec[i] = '0'}
        \}
        \texttt{ctn.modifySpecs(spec.in, readyvec, spec.out, startvec, waitstate)}
      \}
      \texttt{waitstate.add(spec.in, bitvec, spec.out, spec.start, spec.ct)}
    \}
  \}
\}
7.5.8 Generation of ready signals

The controller has one output port (pin) that is used for the controllers ready signal. This pin is connected to the networks ready output node. The world outside the network may now sense when the network has finished its execution. In section 3.3 we have discussed when a ready signal should be generated by the controller. The result was that the ready signal should be generated whenever,

1. $\text{start}$ and the next state is determined to be the standby state.
2. $\text{start}$ and the next state would have been the standby state if $\text{start}$.

In algorithm 7.20, we have translated these conditions for specs in control nodes. The first line of the if condition specifies the first case. The other lines in the if condition specifies the second case.

**Algorithm 7.20** (Generate the controllers ready signal)

```plaintext
CtGraph::generateReady()
{
    /* Pre: this = ctg and nwg is the network graph in the design with the controller ctrl and $p$ is the index of the entry in the outputvectors of the specs corresponding to the ready output pin of $ctrl$ */
    for all (ctn $\in$ ctg.Nodes $\land$ ctn.type = "output" $\land$ state $\land$ spec $\in$ ctn.Specs)
    {
        outvec = makeOutputVector(true)
        if ((spec.in $\Rightarrow$ makeOutputBitVector($\sigma$) $\land$ spec.ns.type = "output") $\lor$
            (spec.in $\Rightarrow$ makeOutputBitVector($\sigma$)) $\land$
            ($\exists$ p $\in$ ctg.Paths $\land$ p.dest.type = "output" $\land$ term $\in$ p.cond.Terms $\Rightarrow$
                spec.in $\Rightarrow$ makeInputBitVector(term)) $\land$
            ($\forall$ : 0 $< i <$ #spec.ready : spec.ready[i] $\neq$ 0'))
        {
            outvec[p] = '1'
        }
        else
        {
            outvec[p] = '0'
        }
        spec.owner.modifySpecs(spec.in, spec.ready, outvec, spec.start)
    }
    end all
}
```
7.5.9 Disable tri-states

The last step before we can copy the spec's into rows of the TAB table, is to disable the tri-states when the control signal to a tri-state is not specified. We need to do this, because we want to be sure that there will not be signal conflicts in the network. Therefore, in algorithm 7.21 we set all '-' entries of the spec's output vectors corresponding to pins connected to the control ports of tri-states on '0'.

Algorithm 7.21 ((Set don't-cares to control ports of tri-states on '0'))

```c
CtGraph::disableTriStates()
{
    /* Pre: this = ctg and nwg is the network graph in the design
       with the controller ctrl */
    for all (ctn in ctg.Nodes \ctn.type = "output"|"state" \ spec in ctn.Specs\  
        port in ctrl.inPorts \ port.edge.dest.type = "tri-state"\ 
        port.edge.destport.name = "control")
    {
        outvec = makeOutputBitVector(port, low)
        ctn.modifyspecs(spec.in, spec.ready, spec.out, spec.start)
    }
    end all
}
```
7.6 Extract Net and Module Lists

In the last process of our interconnect and controller generator, we only have to export the module and netlists representing our register transfer level network. The modulelist is quite trivial to generate, since each node in the network graph corresponds to one network device. However, the "constant" nodes in the network graph may be eliminated by generating the constant signals using hardwiring.

The generation of the netlist is a little bit more complicated (but still trivial), because a network edge could be connected to two ports with different widths. The transformations between bitvectors with different widths (page 2.3.1), may be applied here. This results in six different connection schemes.

1. Connection between two port with same widths (figure 7.3).
2. Unsigned integer or boolean or two-complement integer connection to a destination port with a smaller width than the origin port (figure 7.4).
3. Signed magnitude integer connection to a destination port with a smaller width than the origin port (figure 7.5).
4. Unsigned integer or boolean connection to a destination port with a bigger width than the origin port (figure 7.6).
5. Two-complement integer connection to a destination port with a bigger width than the origin port (figure 7.7).
6. Signed magnitude integer connection to a destination port with a bigger width than the origin port (figure 7.8).

Using these schemes it is kind of straightforward, how to extract the module and netlists from the network graph.

![Figure 7.3: Connection between ports with same width](image)
Figure 7.4: Unsigned integer or boolean or two-complement integer connection, (large to small port)

Figure 7.5: Signed magnitude integer connection (small to large port)

Figure 7.6: Unsigned integer or boolean connection (small to large port)

Figure 7.7: Two-complement integer connection (small to large port)
Figure 7.8: Signed magnitude integer connection (small to large port)
Chapter 8

Results

An interconnect and controller generator has been implemented as part of the NEAT system. The complete package is written in C++, and is divided into 7 programs:

1. **check** - This program checks the input design (sections 2.6, 3.4, 5.4 and 6.4).
2. **prep** - Condition analysis on data flow and control graphs (section 6.2).
3. **mapp** - Map data flow paths onto network edges (section 7.2).
4. **coso** - Solve signal conflicts in the network graph (section 7.3).
5. **cntr** - Create interconnections for control purposes (section 7.4).
6. **tabl** - Extract TAB table from design (section 7.5).
7. **esca** - Generate module and netlists for further processing and graphics (section 7.6).

These programs should be run in this order. The complete interconnect package has also been integrated with the Escape graphical interface.

8.1 Data flow and network conditions

The data flow and network conditions are implemented as a set of bitvectors. However, this is not the best way to implement boolean expressions. Memory can be saved by using bdds\(^1\). It is also much easier and faster to reduce boolean expressions represented in bdds. It is important to reduce the conditions, because the number of productterms used for one condition affects the number of rows used in the TAB table. The less productterms we use to represent a condition, the faster the TAB table generation processes will run, because in many iterations, we iterate over the productterms of conditions.

As one may have noticed, in this report we have already adopted the use of bdd’s in our algorithms. However, the bitvectors are still preferred in the generation of the TAB table.

\(^1\)More information on bdds is available from ir. Geert Janssen, Department of Electrical Engineering, Eindhoven University of Technology.
8.2 Special value nodes in bm constructs

In section 2.4 and 5.3.3, we have seen that sometimes it is advantageous to have the control port of the merge nodes connected to a special value node, while the control ports to the branches of the same bm construct are connected to other ports. But in chapter 6 and 7 we discarded this, for simplicity. Our interconnection process have only to be modified at one point in order to account for the special value nodes in bm-constructs. When a data flow productterm is transformed into its network equivalent (transformToNetworkCondition), we have to look whether there are literals in it, corresponding to pins of condition generating ports that are connected to the input of a special value node. Let's say that there are such literals, and they correspond to a condition generating port owned by $dfn$. In that case we have to know in which state the network condition is going to be used. If it is going to be used in a state $ctn$, with $I(dfn, ctn) = 0$, then we know that $dfn$ is not scheduled in $ctn$ and we have to replace these "unavailable" literals by those corresponding to the output pins of the special data flow node.

8.3 Interconnect optimization

In this report we have not talked about optimization of interconnection in the network. The network generated by our interconnector is called a 1-level mux network. Because a connection between two modules (not multiplexers) passes at most one multiplexer. In our network each multiplexer is connected to one input port of a non-multiplexer device. If we place the multiplexers very close (short wires) to the input port to which it's connected, we will have as many busses (long wires) in the network as there are output ports. This number of busses can be reduced by reusing the same wiring for even more connections. In general this will lead to more interconnection modules, like multiplexers. And the network will become a 2 or higher level network. Notice that a possible interconnect optimizer could be inserted easily between the processes mapp and coso of our package.

8.4 Combinatoric delay and chaining

In this report we have assumed that execution of a combinatoric module does not cross a cycle transition. However, by changing the definition of the delay index for combinatoric devices, we could also process combinatoric modules with delays bigger than the cycletime and execution across cycle transitions. Thus, chaining of combinatoric modules across cycle transitions are also possible with a redefined combinatoric delay index.
Chapter 9

Conclusion

We have developed an interconnect and controller generator, which serves as the last part of the NEAT High Level Synthesis system. We did this by first examining the design representation, consisting of graphs and links. This resulted in the delay index, which turned out to be very helpful in describing the interconnection process. The delay index also gave us the means to prescribe the use of nodelinks in representing scheduling and binding information.

In developing the interconnect and controller generator, we have mainly focussed our attention on the designs without loops in the data flow and control graphs. The network build by our generator will be a 1-level mux network. The interconnect and controller generator has been implemented as a package of subprocesses, such that it can be understood and maintained more easily.
Acknowledgements

I would like to thank Prof. dr. ing J.A.G. Jess for giving me the opportunity to work on my final project in his group, the Design Automation Section of the Eindhoven University of Technology. I thank my mentor Jos van Eijndhoven for his support and patience during the discussions we had on interconnect and controller generation.

I am especially indebted to the other members of the NEAT-club, Ric Hilderink, Marc Heijligers, Wim Philipsen, Adwin Timmer and Harm Aarts for introducing me in the field of high level synthesis. Special thanks to Ric for proofreading this thesis.

I would like to thank the other members of the group for their help and support. Especially I would like to mention Hans Fleurkens, Frans Theeuwen, Geert Janssen and Ed Huijbregts.
Bibliography

    *Optimal Interconnection in Digital Systems Satisfying Concurrency Constraints*
    Int. J. Systems Sci.
    vol. 15, 1984, no. 9, p. 991-999.

    *Data Path Synthesis using Path Analysis*
    Proc. of 28th Design Automation Conference
    p. 591 - 596

[Camposano 87] Camposano R. and J.T.J. van Eijndhoven
    *Combined Synthesis of Control Logic and Data Path*
    IEEE 1998

[Camposano 89a] Camposano R. and W. Rosenstiel
    *Synthesizing Circuits from Behavioral Description*
    IEEE trans. on CAD

[Camposano 89b] Camposano R.
    *Behavior Preserving Transformations for High Level Synthesis*
    Lecture Notes in Computer Science
    vol 408. Springer-Verlag, 1989

[Eijndhoven 91] van Eijndhoven J.T.J et al.
    *The ASCIS Data Flow Graph: Semantics and Textual Format.*
    Eindhoven: Eindhoven University of Technology, Department of Electrical Engineering,
    EUT Report 91-E-251.
[McFarland 83] McFarland M.C. and A.C. Parker
An Abstract Model of Behavior for Hardware Descriptions
IEEE trans. on Computers
vol C-32, no 7, p. 621 - 636.

Tutorial on High Level Synthesis
proc. 25th ACM/IEEE Design Automation Conference

The High-Level Synthesis of Digital Systems.

Data Path Optimization in Modern Computers.
Int. J. Electronics

Petri Net Approach to the Evaluation of Maximum Compatible Classes in Data Path Optimization of Modern Computers.
Int. J. Electronics

[Haver 81] Haver L.J. and Parker A.C.
A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic
IEEE proc. 18th DAC

[Haver 83] Haver L.J. et al.
A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic IEEE trans. on CAD
vol CAD-2, no 1, p. 4 -17.

Pico, logische synthese handleiding.
Eindhoven University of Technology
second edition, dec 1991
[Jong 91] de Jong G.G.
*Dataflow Graphs: System Specification with the Most Unrestricted Semantics*
Proc. of EDAC 1991

[Lee 89] Lee Jiahn-Hung et al.
*A New Integer Lineair Programming Formulation for the Scheduling in Data Path Synthesis*

[Lukkassen 90] Lukkassen R.W.J.
*Register and Multiplexer Minimization in Lowly Multiplexed Data Paths*
Eindhoven University of Technology, Department of Computer Science, 1990.
Master thesis.

[Mathialagan 80] Mathialagan A. and N.N. Biswas
*Optimal Interconnections in the Design of Microprocessors and Digital Systems*

*The NEAT System.*
(Not published).

[Peek 91] Peek J.A.
*Multiplexing of Datapaths: Algorithms and Implementations*
Eindhoven University of Technology, Department of Computer Science, 1991.
Master thesis.

[Rao 89] Rao S.S.S.P and J.R. Isaac
*Interface Optimization: An Algorithm for the Detection of Data Path Redundancy and Reconfigurability towards Obtaining Minimal Bus Interface.*

[Stok 91] Stok L.
*Architectural Synthesis and Optimization of Digital Systems*
[Stok 90a] Stok L.  
*Interconnect Optimization during Data Path Allocation*  

[Stok 90b] Stok L.  
*Interconnect optimization for multiprocessor architectures*  
Proc. of CompEuro 1990, 8-10 May 1990.  

[Stroustrup 91] Stroustrup B.  
*The C++ Programming Language (second edition).*  

[Torng 77] Torng H.C. and N.C. Wilhelm  
*The Optimal Interconnection of Circuit Modules in Microprocessor and Digital System Design.*  
IEEE Trans. on Computers,  

[Tseng 86] Tseng C.J. and D.P. Siewiorek  
*Automated Synthesis of Data Paths in Digital Systems*  
IEEE Trans. on Computer Aided Design,  
vol. CAD-5, 1986, no. 3, P. 379-395

[Tseng 81] Tseng C.J. and D.P. Siewiorek  
*The Modeling and Synthesis of Bus Systems*  