INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Generating a control/data-flow graph representation of a circuit from VHDL for use in circuit synthesis

By David Caron

A thesis submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements for the

degree of:

Masters of Engineering

Department of Electronics Engineering

Faculty of Engineering

Carleton University

January 1998

© copyright

1998, David Caron

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27011-4

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis:

Generating a control/data-flow graph representation of a circuit from VHDL for use in circuit synthesis.

submitted by David Caron in partial fulfillment of the requirements for the degree of
Master of Engineering.

A handwritten signature in black ink, appearing to read 'John P. Knight', written over a horizontal dashed line.

Prof. J.P. Knight

Thesis Supervisor

A handwritten signature in black ink, appearing to read 'J.S. Wight', written over a horizontal dashed line.

Prof. J.S. Wight

Chair, Department of Electronics

Carleton University

January, 1998

Abstract

This thesis demonstrates how a behavioral description of a circuit written in the hardware language called VHDL has been compiled into an internal representation called bdsIf.

This internal representation is based on a series of control/data-flow graphs to capture the behavioral aspects of the circuit. Careful attention has been given to the proper construction of if, switch and loop control constructs based on the behavioral descriptions from the VHDL language. Synthesis tools can be run on the internal representation to generate a register-transfer level description of the circuit. A graphical user interface was designed and constructed to display the control/data-flow graphs in a hierarchical fashion for the benefit of circuit and synthesis system designers. A mouse driven browser allows the designers to browse through the graph to any level of detail.

Acknowledgments

I would like to thank John Knight for making time in his crazy schedule to review my thesis numerous times and also for providing me with funding.

I would also like to thank Spiros Boucouris of Functionality for providing me with a structural VHDL compiler to start off with.

Walter Fergusson, the system administrator for our department, deserves thanks for taking the time to setup a stand-alone unix system to permit me to work uninterrupted from home.

My girlfriend, Geske Bak, deserves much credit for providing me with an incentive to complete this thesis as quickly as possible.

Table of Contents

Chapter 1: Introduction to Synthesis	1
1.1 Master Objective	2
1.2 Contribution	4
1.3 Related work	5
1.4 Layout	6
Chapter 2: An abstract look at behavioral synthesis.....	7
2.1 Levels and Domains of Description.....	7
2.2 Step by step design flow	9
2.3 System-level synthesis	9
2.4 High-level Synthesis	10
2.4.1 Scheduling	12
2.4.2 Resource allocation	14
2.4.3 Resource assignment	14
2.5 Register-Transfer Level Synthesis	17
2.5.1 Data path synthesis	18
2.5.2 Controller Synthesis	20
2.6 Logic Level Synthesis	20
2.7 Technology Mapping	21
Chapter 3: A Silicon Compiler	23
3.1 The behavioral domain description in VHDL	23
3.1.1 What is VHDL?	23
3.1.2 VHDL behavioral descriptions:.....	23
3.2 The In-Core VHDL parser.	29
3.3 The bdsII internal representation.	30
3.4 The Scheduler (Force directed).....	31
3.5 Allocating and assigning functional units.....	33
3.6 The Register-Transfer level.....	36
Chapter 4: The Control/Data-Flow Graph (CDFG).....	37
4.1 The general architecture.....	37
4.1.1 Nodes	37
4.1.2 Terminals	37
4.1.3 Edges	38
4.1.4 Hiernodes.....	38
4.1.5 Port nodes	38
4.2 The data-flow	39
4.2.1 Read write node style	40
4.2.2 Pure data-flow style	41
4.3 The control-flow.....	41
4.3.1 The fork node	41
4.3.2 The join node.....	42
4.3.3 Types of control-flows	43
4.3.4 The if construct.....	44
4.3.5 The switch construct.....	46

4.3.6	The loop construct	49
4.3.7	Wrapping up a CDFG.....	51
4.3.8	CDFG Hookup.....	51
4.4	Displaying CDFG's.....	52
4.4.1	The postscript generation utility	52
4.4.2	A crossover reduction drawing algorithm with unique edge colors.	54
4.4.3	Reducing crossovers through improved ordering of nodes and terminals	54
4.4.4	An algorithm for node placement during CDFG drawing.....	55
Chapter 5: A real example: "The traffic light"		57
5.1	The specification	57
5.2	System level synthesis	58
5.2.1	The controller	58
5.2.2	The highway traffic signal	59
5.2.3	The farm road traffic signal	60
5.2.4	The short timer.....	61
5.2.5	The long timer	61
5.3	High level synthesis	61
5.4	The XbdsII executable.	73
5.4.1	Loading.....	74
5.4.2	Viewing CDFG's	75
5.4.3	Exiting	75
5.5	The Hierarchy Structure.....	75
5.5.1	The test bench.....	76
5.5.2	The Implicit architectural loop	76
5.5.3	The Architecture Body	77
5.5.4	VHDL statement encapsulation.....	80
5.6	Register transfer level synthesis.....	80
Chapter 6: System Implementation		81
6.1	Platform.....	81
6.2	OverView	81
6.3	In-Core Classes	85
6.3.1	The parser	85
6.3.2	The VAST classes.....	85
6.3.3	The ICvhdlParser class	86
6.3.4	The ICbase classes.....	86
6.3.5	Type classes	89
6.3.6	Limitations.....	91
6.4	Implementation details.....	92
6.5	Overview	92
Chapter 7: Conclusions		95
7.1	Summary	95
7.2	Innovations.....	96
7.3	Benefits	97
7.4	Future improvements	97
Appendix A:Related Work.....		100
7.5	A data-flow only model	100

7.6	Sequencing Graphs	100
7.7	Value Trace (VT)	101
Appendix B: The history of BDS		103
Appendix C: The X Window system.		105
C.1	Xlib	105
	C.1.1 Setup	106
	C.1.2 Drawing	108
C.2	X Toolkit IntrinsicS	109
C.3	Widget Sets	109
C.4	X Window Manager	110
Appendix D:Implementation details		111
D.1	The bdsII parts that generate CDFGs.....	111
	D.1.1 The Element directory	111
	D.1.2 The graphContainer directory.....	112
	D.1.3 The CDFGData directory	113
	D.1.4 The CDFG toolkit.....	114
	D.1.5 The construct toolkit.....	115
D.2	The graphical elements	118
	D.2.1 The Vertex.....	119
	D.2.2 The Arc	121
	D.2.3 The Graph.....	123
D.3	The front-end Graphical User Interface (XbdsII)	126
	D.3.1 The application builder.....	126
	D.3.2 The external interface file	129
Appendix E:Building a CDFG using toolkit calls		131
E.1	Initialization	131
E.2	The root CDFG	131
E.3	Declaring variables	131
E.4	Creating a sub CDFG.....	133
E.5	The Parser Variable Table.	133
E.6	Creating Nodes.....	133
E.7	Creating edges.....	135
E.8	Connecting Nodes and Edges	136
E.9	Making the association between an edge and its variable.	136
E.10	Updating entries in the ParserVT.....	137
E.11	Wrapping up a cdfg.....	137
E.12	Generating a postscript file	138
E.13	Creating behavioral constructs.....	138
	E.13.1 The if construct.....	138
	E.13.2 The switch construct.....	139
	E.13.3 The loop construct	139
E.14	Generating a hiernode of a CDFG	140
E.15	Termination	141
Master index		142

List of tables

TABLE 1.	State transition table for the finite state machine of figure 8.	17
TABLE 2.	Critical path operation to functional unit assignment	33
TABLE 3.	Mobile operation to functional unit assignment.	34

List of Illustrations

FIGURE 1. A structural description of a circuit.	1
FIGURE 2.b One possible register-transfer level diagram.	3
FIGURE 2.a A data-flow graph with <i>precedence edges</i>	3
FIGURE 3. The universal Y-chart for circuit design.	7
FIGURE 4. Conversion processes between domains of the Y-chart in figure 3	9
FIGURE 5. The steps involved in high-level synthesis	10
FIGURE 6. A scheduled data-flow	13
FIGURE 7. Resource limited scheduled data-flow diagram.....	15
FIGURE 8. A register-transfer level diagram based on the data-flow diagram in figure 6 on page 13.	16
FIGURE 9. A Moore versus Mealy machine.	19
FIGURE 10.a Slow FIR filter.....	19
FIGURE 10.b Fast FIR filter.....	19
FIGURE 11. A graphical view of the VHDL counter example.	27
FIGURE 12. A typical framework of a VHDL program.	28
FIGURE 13. As Soon As Possible scheduling verses As Late As Possible scheduling....	32
FIGURE 14.a The unbalanced distribution graph of probability verses control step, separated by operation.	33
FIGURE 14.b The balanced distribution graph of probability verses control step, separated by operation.	33
FIGURE 15. An operation to function unit cross-reference table for operation assignment.....	35
FIGURE 16. An example of a basic control/data-flow graph.....	38
FIGURE 17. The framework of a hiernode.	39
FIGURE 18. An example of the read write node style.	40
FIGURE 19. The fork node.	42
FIGURE 20. A two input flow join node.	42
FIGURE 21. A generic if construct without an else.	45
FIGURE 22. A generic if construct with all control-flow types except for goto.	46
FIGURE 23. A generic switch construct with normal and break control-flow.....	48
FIGURE 24. A generic loop construct with top and bottom body simultaneously.	50
FIGURE 25. A postscript drawing of a CDFG.....	53
FIGURE 26. To bend or not to bend.....	55
FIGURE 27. A carelessly drawn CDFG.....	56
FIGURE 28. The top level diagram showing the signal flow between the five processes.	59
FIGURE 29. The finite state machine for the controller process.....	60

FIGURE 30. Signal initialization for the architecture body represented in topBody.	63
FIGURE 31. A condition CDFG consisting of an AND operation	64
FIGURE 32. Another condition CDFG.	65
FIGURE 33. An unary operation CDFG	65
FIGURE 34. A direct assignment CDFG.	66
FIGURE 35. A complete if statement CDFG.	66
FIGURE 36. A switch CDFG	67
FIGURE 37. This CDFG represents the wait condition of the process labelled Controller_Process.....	68
FIGURE 38. A CDFG of a conditional signal statements shown below	69
FIGURE 39.a. The condition CDFG from a for loop.	70
FIGURE 39.b. The bottombody of a for loop.....	70
FIGURE 40.a. The CDFG of a complete for loop.....	71
FIGURE 40.b. This CDFG represents the process labelled. Long_Timer.....	71
FIGURE 41. This CDFG represents an instantiation of the entity/architecture labelled traffic_light_controller	73
FIGURE 42. The graphical user interface for bdsII.	74
FIGURE 43. This diagram represents the variable dependencies between all the architecture statements in the entity/architecture labelled traffic_light_controller	78
FIGURE 44. The wait if construct.	79
FIGURE 45. This CDFG represents the contents of the then hiernode from figure 44 on page 79.....	79
FIGURE 46. User triggered events.	82
FIGURE 47. Levels of abstraction from the GUI down to the Xlibrary functions.....	83
FIGURE 48. A diagram of the three packages which together constitute bds97.	84
FIGURE 49. A model of the memory image of the In-Core base class structure.	87
FIGURE 50. Hierarchy of In-Core data type classes.....	90
FIGURE 51. The Hierarchy of In-Core expression data classes.	91
FIGURE 52. Calling sequence for the open, input mouse events and exit callback functions.....	94
FIGURE 53. A sequencing graph demonstrating a conditional hiernode and a multiple execution thread	102
FIGURE 54. A graphical representation of the CDFGData class.	113
FIGURE 55. The CDE application builder.....	127

Abbreviations

ALAP: As Late As Possible

ALU: Arithmetic Logic Unit

ASAP: As Soon As Possible

ASICs: Application Specific Integrated Circuits

CDE: Common Development Environment

CDFG: Control/Data-Flow Graph

GNU: GNU Not Unix

GUI: Graphical User Interface

FFT: Fast Fourier Transform

FSM: Finite State Machine

ROM: Read Only Memory

RTL: Register Transfer Level

SIF: Sequencing Intermediate Form

VHDL: VHSIC Hardware Language

VT: Value Trace

Glossary

architecture

In the context of VHDL, the architecture block defines the parts of a component in full detail, in terms of component instantiations, concurrent statements, and sequential processes.

callback function

In the context of the XWindow System, a callback function is one which is automatically invoked by a widget during a specific event, such as when a widget button is pressed or a widget menu item is selected.

concurrent, concurrently, concurrency

Objects which are concurrent are said to be executed at the same time or in parallel, as opposed to one at a time.

control-flow

An particular path in graph which determines which operations will be executed, as well as the order of execution of the operations.

control step

A discrete point in a control-flow which determines the operation or operations which are to be executed concurrently. Operations pertaining to the next control step may not start until those in the current control step have executed until completion.

data dependencies

A data dependency exists between two operations executed one after another if the latter operation requires a value which is computed by the former operation.

data-flow

The direction in which data-flows from the output of one operation to the input of another.

determinant

The determinant is an algebraic operation used on a matrix to determine, for example, if an inverse of the matrix can be found. If each row in a matrix represents a point in space, then the sign of the determinant of the matrix will specify the orientation of the points with respect to each other.

encapsulation

Creating a self contained function where coding details can be hidden behind a common calling interface which can be used by many objects.

entity

In the context of VHDL, an entity block declares a component's interface to the outside world. The interface consists of input ports, output ports, and possibly bidirectional ports called inout ports.

frontier

A list of coordinates, used as a framework for determining where to place Vertices with respect to each other onto a pixel map. The pixel map is divided up into a grid of boxes. Each box can hold one Vertex. Each entry in the list represents a row in the grid. The coordinates at that entry represent to next available column in that row.

functional unit

This term defines a component at the register-transfer level. Adders, ALU's, comparators, etc... are all examples of functional units.

hangers

Hangers are programming structures which allow new objects to be referenced by existing objects without altering the existing object. The new object is "hung" onto the existing object through a cross reference list. The existing object can access the new object by finding an entry of itself in the list and then grabbing the cross-reference pointer.

hiernode

In a control/data-flow graph, a hiernode is a node which itself contains a control/data-flow graph, whose imports and outputs are implicitly connected to the input and output terminals of the hiernode. A hiernode is an instance of a control/data-flow graph in the sense that multiple hiernodes may represent the same CDFG.

internal representation

Refers to structure used to store a CDFG hierarchy in memory.

lexical analyzer

This is a program which reads in an input text file containing a context-free grammar, and converts it, character by character, into a sequential list of syntactic groupings which are made up of tokens. A reserved word in a programming language such as "then" would be converted into a token. A subsequent program would then parse the tokens to produce a syntactic grouping such as an if statement.

list scheduling

A way of scheduling operations in a data-flow, dependent on resource constraints.

Mealy Machine

A finite state machine whose outputs are not only determined by the current state of the machine. but also by the inputs to the FSM.

minimization

In scheduling, the act of reducing the total number of control steps, or the total number of resources, at the expense of the other.

Moore Machine

A finite state machine whose outputs are only determined by the current state. The inputs serve only to determine the next state of the machine.

multi-cycling

In scheduling, some operations such as multiplying may take substantially longer than most other operations. Instead of increasing the control step time to allow the long operations to complete, two control steps are allocated to one long operation.

net list

A structural description of a circuit. Each line in the list describes the nets connected to one specific component.

pixel map

A structure which represents the way colors are defined on the physical display screen. The structure consists of a table of rows and columns where each entry represents the color of a screen pixel by an index into a color palette. Each entry in the color palette defines the actual color.

postscript

A popular text based format for sending data to printers.

precedence edge

Used to indicate which operations will be executed in a data-flow graph depending on the results of a compare operation.

resource

In scheduling, a resource is a hardware unit which executes an operation. Similar operations which are scheduled in different control steps may be able to share the same resource.

resource constrained

Given a limited number of resources, optimize the number of control steps

schedule

In a data-flow graph, a schedule determines the allocation of operations to control steps.

sequential

Indicates that operations are to be executed one after another, in series.

storage element

A buffer or register file which can store the results of operations between control steps.

syntactic grouping

A program statement as it appears in the text of source code. Ex. IF C==1 THEN F=A+B
ELSE F=A-B.

time constrained

Given a specific number of control steps, optimize the number of resources.

verilog

A popular hardware description language

widget

A graphical component in XWindows which has its own appearance and behavior when handling mouse, keyboard, and other system events. The action taken by a widget is determined by the contents of callback functions which are automatically invoked by the widget.

Chapter 1: Introduction to Synthesis

Circuit synthesis was born out of a need to design circuits at a higher level of abstraction, compared to traditional design levels. Today's circuits, filled with lots of new and interesting features, must be faster, smaller in size, and use much less power than the previous generation. The most efficient way to achieve all of these design constraints has been to try and compress the entire circuit onto one chip. Circuit simulation has allowed high speed designs which take propagation delays into account. New technology has reduced chip power consumption. However, market demands dictate that still more features are needed, which must be produced at a lower cost and should be ready for production yesterday. As a result, circuit design at the logic level has continued to become more complicated, reducing the time to market and driving up design costs. By synthesizing high-level circuit descriptions, complex circuits can be automatically designed much faster at a reduced design cost. Such computer generated designs are bound to be larger, slower and absorb more power, however with current technology, these limitations can be minimized.

This thesis will concentrate on behavioral synthesis, which describes a circuit algorithmically as opposed to structurally. A structural description of a circuit amounts to a list showing the connections to the circuit, which can be entered textually or graphically. This is called a *net list*. Figure 1 contains an example of each.

FIGURE 1. A structural description of a circuit. The signals A and B are four bit inputs while the signal F is a four bit output.

An algorithm consists of blocks of sequential steps, some of which may only execute depending on a condition. Computer languages such as Pascal and C can be used to describe algorithms for anything from tic-tac-toe to how a miniature robot finds its way through a maze. A behavioral description of the circuit in figure 1 on page 1 is presented as follows:

```
if C=1 then F=A+B
    else F = A-B;
```

From this basic example it can easily be seen how much simpler the behavioral description is not only to generate but also to modify. At the high-level, behavioral synthesis will compile a behavioral description of a circuit specification into an internal representation, which can be displayed as a control and data-flow. This in turn will be converted into a register-transfer level (RTL) description. An RTL description describes the connections required between chosen hardware components in order to implement the circuit. These components consist of *functional units* to execute operations, and registers to store intermediate and final results from those operations. In addition, a finite state machine is usually needed to coordinate the operations. An example of a data-flow of the above behavioral description is presented in figure 2.a. The dotted lines represent *precedence edges* which determine which operation will be executed dependent on the outcome of a compare operation. Figure 2.b. represents a RTL diagram which might be generated from the data-flow. As shown, the output of the compare *functional unit* directly drives a multiplexor. In a more complicated RTL diagram, a finite state machine may be required to control all the multiplexors and *functional units*. An example of such an RTL diagram is shown in figure 8 on page 16.

1.1 Master Objective

This thesis is a part of a master objective, that of creating a synthesis system for digital circuits. To that end, the implementation of the system is based on two components. The first is a compiler to parse a behavioral description written in the VHSIC Hardware Description

FIGURE 2.a A data-flow graph with *precedence edges*. The addition operation is executed if $c=1$. Otherwise the subtraction operation is done instead. A data join ensures that the result is assigned to one F.

FIGURE 2.b One possible register-transfer level diagram. It could also have been implemented using a functional unit which performed both addition or subtraction since, as the data-flow graph demonstrates, the operations are *mutually exclusive*.

Language (VHDL). The second is an intermediate format which consists of a hierarchy of Control Data-Flow Graphs (CDFGs). Each of these components will be briefly described:

The VHDL compiler: This component will open and parse a VHDL source file. The compiler, which is a part of a system called In-Core, was provided for this system by Spiros Boucours of Functionality. It was only designed to fully compile VHDL structural statements. It had to be extended to compile behavioral statements.

bdsII: This component is a merger of work started at Carleton University and Nortel.¹ BdsII consists of library toolkit functions which can be invoked by another program, such as a compiler. The toolkit functions allow CDFGs to be generated in a hierarchical manner. This software was provided by Nortel on an “as is” basis. High level behavioral constructs such as; if branches, switches and loops were documented conceptually, but were not implemented. A limited postscript file generation utility allowed very simple CDFG’s to be printed.

1. Credit for bdsII is owed to Trevor May, Francis Langlois, Arthur Castongany, Pierre Paulin, Christopher Donawa, Yatish Kumar, Shailesh Sutarwala, and Cliff Liem. See Appendix B on page 103 for details.

1.2 Contribution

It is by no means clear how software algorithms should be compiled into hardware. One of the difficult things about developing new algorithms is visualizing the data-flow graphs and their development into circuits. A consistent method for implementing control constructs like if, switch and loops has been implemented. Due to the complexity of the resulting internal representation, a graphical representation was needed which could be generated quickly to verify the correctness of the generated control constructs. The layout of the internal representation, as a hierarchy of CDFGs, allows larger, control intensive, circuit designs to be displayed graphically while maintaining readability. The hierarchy can further be used to hide or expose details according to the needs of the compiler designer. It should be noted that a graphical browser was deemed necessary to properly analyze and refine the model used to generate the control constructs.

This thesis has contributed to the main objective by designing and implementing a user friendly graphical browser. The browser aids in the development of synthesis programs by rapidly visualizing an *internal representation*. The *internal representation* will store information as a hierarchy of modules, which represent the structural and behavioral parts of a circuit description. The browser allows a designer to see their software algorithms at a glance, before they are compiled into hardware. This will give the designer an opportunity to refine and optimize the algorithms.

This contribution has been made to the master objective through the following steps.

- High level behavioral constructs have been implemented in bdsII. These constructs are described in chapter 4.
- The In-Core VHDL compiler has been upgraded to handle behavioral concepts. These are described in section 6.3.4.
- The main parser class of the In-Core compiler, which is responsible for generating an internal representation from the VHDL source code, has been vastly augmented by two set of bdsII toolkits function calls. One set allows CDFG's to be built, one element at a time while the other set connects CDFG's together to form behavioral constructs. The toolkits themselves are described in section D.1.4 and section D.1.5 of appendix D. An example of how the toolkit functions have been used in the main parser class to generate a hierarchy of CDFG's is presented in appendix E.

- A set of three additional classes have been added to the bdsII internal representation to setup a drawing pane, define colors and then draw all the elements of a CDFG onto the screen. The set of three classes, called graph classes, are described in section D.2 of appendix D.
- The order and organization of inport nodes, outport nodes, and hiernode terminals created by the bdsII toolkit functions have been carefully sequenced to produce a well ordered internal representation, which not only produce aesthetically pleasing graphs when the CDFGs are drawn into the screen, but will also simplify the implementation of an output stream to be read in by another program.
- A user friendly graphical application, called XbdsII, has been designed and implemented under the Common Development Environment (CDE). This program provides the front-end graphical user interface for the In-Core parser and bdsII graph classes. XbdsII is described in section D.3 of appendix D.

1.3 Related work

The work described below provides examples of how control/data-flow graphs are being used for scheduling and synthesis testability. Examples of other methods proposed for representing behavioral constructs is presented in appendix A.

A conference paper by S. Amellal and B. Kaminska[1] presents a model for generating a single data and control flow graph to represent a VHSIC hardware description language (VHDL). The VHDL control constructs can be implemented as conditional dependency edges similar to that shown in figure 2.a. The mutual exclusion between nodes, such as the addition and subtraction node in figure 2.a, is recognized and can then be appropriately handled by a scheduler.

A journal paper by Lin Chi-Ho and Chong Jong-Wha[2] describes a scheduling algorithm which takes advantage of the resource sharing inherent in mutually exclusive nodes produced by the conditional branches of a CDFG. In this paper, the intermediate data structure (CDFG) is generated by a VHDL analyzer. The scheduler transforms a CDFG with conditional branches into a scheduled CDFG without conditional branches.

A paper by T.C. Lee, N. K. Jha and W. H. Wolf[3] makes use of the term HCDFG which stands for Hierarchical Control/Data-Flow Graph. They are used to model system behav-

ior which has been designed with built in testability. A post-order traversal of the HCDFG is used during testability synthesis in an effort to reduce the number of additional registers which arise due to the testability embedded in the circuit description. The method used was found to generate fewer additional registers when compared to some other design-for-test methods.

Another method to improve synthesis testability has been discussed in a paper by F. F. Hsu, E. M. Rudwick, and J. H. Putel[4]. Test statements are inserted in behavioral descriptions and control points are added at branch conditions. The generated CDFG can be analyzed to identify hard-to-control loops.

The concept of applying visual languages in the field of hardware design was presented by Wang Qingsheng, Xue Hongxi, Su Ming, and Bian Jinian[5]. They describe an algorithm which generates VHDL code from a graphical description of a circuit using CDFGs. The concept was named visualCDFG.

1.4 Layout

The general layout of this thesis is described as follows.

- Chapter 2 describes in general, all the levels of circuit synthesis from the system level down to technology mapping.
- Chapter 3 follows the same format but a specific example at each level is developed.
- Chapter 4 describes the bdsII control/data-flow graph model in detail.
- In Chapter 5, the final product is demonstrated by synthesizing a VHDL circuit description into a hierarchy of CDFGs.
- Chapter 6 covers some of the implementation details of the In-Core, bdsII, and XbdsII packages.

Chapter 2: An abstract look at behavioral synthesis

This chapter presents an overview of behavioral synthesis, level by level. Initially a description of how behavioral synthesis fits within overall circuit design will be presented.

2.1 Levels and Domains of Description

Partitioning circuit design into many levels and domains allows designers to concentrate on one level of design at a time. The Y-chart [6] in figure 3 on page 7 provides an idea of where high-level design resides with respect to other design levels. The chart also demon-

FIGURE 3. The universal Y-chart for circuit design. .

strates the correlation between the three different domains in circuit design. They are described as follows:

- **Behavioral Domain:** This domain deals with what the circuit will do, independent of how it is implemented. Taking into consideration technological limitations, circuits can be implemented electronically, hydraulically, biochemically. etc... Behavioral descriptions are typically described textually.
- **Structural Domain:** This domain describes components and interconnections. It uses adders, registers, multiplexors, nets, etc... The actual technology CMOS, BiCMOS, GaSi, etc... becomes relevant at the lower levels.
- **Physical Domain:** This domain describes how the circuit will be laid out. The layout is technology dependent at a higher level in this domain.

All three domains can be described at different levels:

- **Circuit level:** This is the lowest level that is commonly used in circuit design. The transistor can be described by its differential equations which calculate its current throughput with respect to its base voltage; it can be represented schematically; or its physical appearance can be drawn graphically.
- **Switch level:** This is an intermediate level where transistors, including their implicit resistors and capacitors, are treated as switches.
- **Logic level:** This is the most common design level. The behavior of logic gates can be described with Boolean algebra; they can be drawn using schematic symbols inter-linked together, or displayed as Standard-Cells.
- **Register Transfer level:** This is the most common level for circuit descriptions generated by current synthesis tools. The circuits can be described by value transfers between registers, or drawn using block schematic symbols.
- **Algorithmic level:** Circuits can be described using behavioral constructs (described later), drawn using black box schematic symbols, or displayed as one block of a chip.
- **System level:** This includes the entire circuit being designed. It can be described in a hardware description language, drawn as a block with a brief description of its operation. i.e. an FFT, or displayed as a chip with I/O pads for the outside world. The input / output requirements are usually described at this level to complete the system specification.

An unscheduled control/data-flow graph fits in between the algorithmic and register-transfer levels in the behavioral domain.

The act of changing from one level to another or one domain to another is called a transition. The transition from a low level to a high one is called abstraction while the opposite is called refinement. Figure 4 demonstrates the transitions between the domains.

FIGURE 4. Conversion processes between domains of the Y-chart in figure 3.

For the remainder of this thesis only the synthesis transition will be developed.

2.2 Step by step design flow

Most if not all designs start with a system specification and end with a complete layout in a chosen technology, ready for fabrication. The two main transitions involved in this process, as shown in figure 4, are synthesis and generation. The latter is better understood and software is readily available and in use to generate a physical layout from a structural description. The former is less understood and the little software available has limited functionality. The next five sections will describe the steps involved to achieve a synthesis transition.

2.3 System-level synthesis

The circuit at the system level refers to the circuit as a whole, including a well-defined interface to the outside world. The system is the solution to the problem which was assigned to the designer in the first place. As of yet there are no good automatic methods available to partition the problem into a set of subproblems. This must be done manually by the designer. These subproblems can be solved by a set of subsystems which can be abstracted as a set of processes which operate *concurrently* with each other. The behavior of these processes can be described *algorithmically*. Each process, once defined, can then be synthesized.

2.4 High-level Synthesis

This type of synthesis consists of parsing a behavioral description of a process, converting it into an internal representation suitable for processing, and then producing a register-transfer level description of an equivalent digital circuit in the structural domain. Figure 5 demonstrates the steps involved.

FIGURE 5. The steps involved in high-level synthesis.

Processes are typically described textually using a language such as VHDL. The input file containing a set of related processes constitutes a system level description. The internal representation consists of a *data-flow* and a *control-flow*. Each of these terms will be explained briefly.

A *data-flow* consists of vertices which represent operations and edges which represent *data dependencies* between the operations. An example of a simple *data-flow* is shown in

the model in figure 6 on page 13. Four data-flows can initially be seen at the top of the model. Two of them are constants and the remaining two are variables. The values in the variables have been previously defined elsewhere while those in the constant are hard-coded or built into the hardware. Edges are used to transfer the values to operations which are represented as vertices in the model. New *data-flows* are then generated which carry the results of the operations onward to other operations or eventually onto external operations outside the scope of the figure. The order in which these operations may be executed is constrained by the *data dependencies* between the operations. The multiplication operation must wait for one of its input values to be computed by the addition operation. The multiplication operation is therefore dependent upon the addition operation and thus they must be executed one after the other or in other words, *sequentially*. On the other hand, the comparison and addition operation do not depend on each other and so they can be executed at the same time, or in other words, *concurrently*.

A *control-flow* is used to represent the order in which operations in a data-flow are executed. Each of the operations in figure 6 on page 13 can be considered to be bound to an invisible control edge starting at the top of the figure and travelling down in the direction of time. A *control-flow* can also represent a path of execution. The model in figure 6 only shows one possible path, however a model for visualizing alternate pathways will be explored in section 4.3.1 The output *data-flow* from the comparison operation in figure 6 can be used to select the desired path or *control-flow* through a branch.

Two main types of designs, suitable for high-level synthesis, have emerged:

- General purpose processor & Digital Signal Processing: These are data-path intensive designs with usually one control path. Trade-offs can be made between the area needed to implement *concurrent* operations versus the total time needed to execute them. A hardware implemented Fast Fourier Transform is an example of such a design.
- Control dominated designs which contain many branches and significant signal processing. Both the control branches and data-flows can be handled efficiently by a synthesis program. Examples of such designs are programmable CD players or programmable VCRs.

Designs which consist only of control branching can be treated as finite state machines. Such designs typically require synthesis by logic minimization, not high-level synthesis.

2.4.1 Scheduling

Once the internal representation has been generated, each vertex or operation in a data-flow must be assigned a time when it will be using a *resource* to perform its function. A *resource* represents a hardware component which is capable of carrying out the operation. For example, an Arithmetic-Logic Unit can be used to perform the comparison and addition operations in figure 6 on page 13 while a hardware multiplier can carry out the multiplication operation.

Operations which are data dependent will be separated by at least one step such that the dependant operation will execute at least one time step after the first operation. An entire graph of vertices or operations will be separated into such steps, based on their *data dependencies* as well as a scheduling algorithm. The steps are formerly called *control steps* because a finite state machine (FSM) controller is used to control the timing. There is often a one-to-one mapping between *control steps* and FSM states. Operations assigned to a particular control step only execute when the controller is in the corresponding state. The physical time delay between *control steps* is fixed and must be long enough to allow the longest operation to execute. The result of each operation is normally stored in a register, or latched at the end of a control step.

It is possible however to assign a very slow operation to two control steps. In this case the results of the operation would not be latched until the end of the second control step. This is called *multi-cycling*. Given sufficient time, a fast operation could be appended to the slow operation within the two control steps. The data-flow between the operations would not be latched in this instance because the slow operation will not have a result within one control step.

The data-flow segment from top to bottom with the longest propagation delay is called the *critical path*. This is represented by the heavy edges in figure 6 on page 13. The other

properties described above are also demonstrated in the figure. The scheduled data-flow is taken from the behavioral description shown below:

FIGURE 6. A scheduled data-flow. If the condition $2 < C$ is false, then both data joins at the bottom of the graph will pass the new computed value of C and P . If the condition is false however, the old values of C and P will be passed through unchanged. The data-flow is inside of a loop. Therefore the values of C and P at the end of one loop must be propagated into the top of the next loop. This can be shown in one of two ways. The output values can be connected with the input values using the dot-dash edges as shown on the left, or the loop body can be duplicated as shown on the right. The right side of the join at the top of the left diagram allows the initial values of C and P to be set to 0 and 1 respectively.

```

c=0, p=1;
while (c < 2)
begin
    c = c + 1;
    p = p * c;
end

```

There are three classes of scheduling objectives:

- *Resource constrained* (list scheduling): Given some specific constraints on *resources*, optimize the number of control steps without mapping two operations in the same control step to the same hardware.
- *Time constrained*: Given some specific time constraint, optimize the use of *resources*, by distributing the operations over as many control steps as possible.
- *Minimization*: An attempt is made to minimize the time, at the expense of *resources*. Or an effort is made to minimize the *resources* at the expense of time. The later can result in a completely serial computation. Most algorithms will also try to minimize some cost functions of time & resources.

An example of a time constrained scheduling algorithm will be described in chapter 3.

2.4.2 Resource allocation

Some scheduling algorithms will allow as many resources as needed to meet a time constraint while other algorithms will backup or restart the scheduling process a different way if one or more types of resources are exhausted. The three different types of resources which must be allocated are:

- *Functional units*: These are hardware units which are used to implement scheduled operations. Individual units may be multi-purpose in which case they can execute many different types of operations (i.e. an Arithmetic-Logic Unit), or they may be dedicated to performing only one type of operation.
- *Storage elements*: At the end of a control step, the data generated from a *functional unit* must be buffered so that it will be preserved for the subsequent control step. A storage element such as a register or a register file will accomplish this.
- *Interconnections*: The values generated by *functional units* are propagated to storage elements through buses. If the results of two or more functional units are to be stored into the same storage element (at different times), then a multiplexor must be used to make the right connection at the right time. A controller net determines the switch state of each multiplexor via nets. The controller net is typically implemented using a finite state machine which coordinates all of the net outputs. Figure 8 on page 16. contains an example of how the controller net connects to the multiplexors.

2.4.3 Resource assignment

To determine if a particular schedule will meet resource constraints, each operation must be assigned to resource instances such that each instance is assigned to at most one operation per control step, without exceeding the number of resource instances of that type. A

good scheduling algorithm will attempt to balance the usage of *functional units* such that one type of functional unit is not exhausted prematurely. If resource constraints cannot be met, then the resource allocation must be increased, or else one control step must be split into multiple steps to allow existing functional units to be assigned to more operations. For example, the scheduled data-flow in figure 6 on page 13 assumes that a comparator and an adder are available. If only one of these resources is available in a control step (i.e. they are implemented in an ALU), then another ALU or adder or comparator must be added. If this solution is not desirable, then the first control step can be broken into two control steps as shown in figure 7 on page 15.

FIGURE 7. Resource limited scheduled data-flow diagram. Because the compare and add operation are in different control steps, both operations may be implemented with the same functional unit. The loop has been omitted from the data-flow for simplicity.

Once all operations have successfully been assigned to *functional units*, then values generated from the instances must be assigned to storage elements. Each storage element can be assigned to at most one value per control step. Attempting to reuse storage elements as much as possible may increase the complexity of interconnections. If the physical dimensions of the final layout is a factor, then busing and multiplexors may require more area than a few extra storage elements. The complexity and size of the controller net is also proportional to the number of multiplexors used.

When all values have been assigned to storage elements, then the link from the functional units must be assigned to buses and multiplexors. Busing area can be minimized by grouping functional unit instances and their registers close to each other, and then using only a few long distance bus lines.

Once all scheduling, resource allocation and assignment has been performed, the final result is a register-transfer level structure which is usually generated in the form of a *net list* in a textual document. An example of a register-transfer level diagram is shown in figure 8 on page 16. This diagram is generated from the data-flow schedule in figure 6 on page 13

FIGURE 8. A register-transfer level diagram based on the data-flow diagram in figure 6 on page 13. Note the use of the B register to save the old value of C in the event that the loop ends. Registers are only written to when their enables are high. The left side of a multiplexor is selected with a 0 on the select and the right side is selected with a 1 on the select.

TABLE 1. State transition table for the finite state machine of figure 8.

Operation(s)	This State	c	b	a	Next C<2=0	State C<2=1	YP	YC	Store_P	Store_C	Store_B
Load P Load C	0	0	0	0	1	1	1	1	1	1	0
B=C C<2 C=C+1	1	0	0	1	4	2	X	0	0	1	1
P=P*C	2	0	1	0	3	3	0	X	1	0	0
P=P*C (cont.)	3	0	1	1	1	1	0	X	1	0	0
WAIT	4	1	0	0	4	4	X	X	0	0	0

The FSM in figure 8 on page 16 consists of four states as shown in table 1. The external values of P and C are loaded in state 0. The next state is unconditionally state 1 where the value of C will be compared to 2. If C is greater than 2, then the loop exits immediately without altering the value of C or P¹. This is shown by placing the FSM into a wait state until it is reset or restarted externally. If C is not greater than 2, then the new value of P is calculated in the next two states and then the loop restarts.

If a register-transfer level diagram were generated for the data-flow schedule in figure 7 on page 15 instead, then the adder and comparator would be replaced by an ALU and one additional state would be needed to separate the addition operation from the compare operation. The following additional signals from the FSM would be necessary to allow the ALU to be used as both an adder and a comparator.

- ALU: This signal is needed to select the operation, 0 for addition or 1 for comparing.
- YV: This selects the right value for the ALU using an additional multiplexor, 0 for the value 1, and 1 for the value 2.

2.5 Register-Transfer Level Synthesis

This level of synthesis accepts a *net list* for a register-transfer level description of data-flow and a state transition graph of a controller. Each part is then handled separately.

¹. The old value of C is saved in register B. The output of register B is then assigned to an external C, so effectively C is unchanged.

2.5.1 Data path synthesis

Data path synthesis involves optimization using the following methods:

Operator Selection:

The actual hardware module chosen from the component libraries depends on the speed requirements of the functional unit. A fast, area intensive module will be used if the functional unit is part of the critical path, otherwise a slow, area efficient module will be used instead. For example, a fast complicated parallel adder would be used in the critical part of the circuit over a slow, bit serial adder.

Mutually exclusive transfers:

If there is a branch in the control structure, only one branch will execute at a time. Therefore the same hardware module can be used in both branches. In fact, they may both be placed in the same control step, thus eliminating a step. A two way, half duplex radio is an example of some of the same circuit components being used to perform two different functions, that of transmitting and receiving.

***Mealy/Moore* automaton:**

In certain circumstances where the operations performed in a then and else part of a branch consist of assignments to the same variable, the multiplexor determining what is assigned to the common variable can be driven directly by the data comparison used for the branch, thereby removing the branch from the controller. Bypassing the clock driven controller also eliminates the one clock cycle delay imposed on the result of the comparison. The then and else states can be merged into one, reducing the complexity of the controller. This effectively reduces a *Moore* automaton into a *Mealy*. An example of this is illustrated in figure 9 on page 19.

Retiming:

The period of the clock cycle is determined by the greatest propagation delay between storage elements. (i.e. registers or latches). If the greatest propagation delay could be

Moore Machine.

Mealy Machine.

FIGURE 9. A Moore versus Mealy machine. The output from the compare operation is immediately transmitted to the multiplexor of the Mealy machine, thereby avoiding the state transition delay imposed by the FSM.

reduced by a rearrangement of storage elements, then the clock period could be further reduced. Figure 10.a contains a data-flow diagram of a small finite impulse response (FIR) filter. A new sample is supplied at the input every clock cycle. The greatest propagation delay consists of both adders. Figure 10.b shows an equivalent FIR filter whose storage elements have been re-arranged such that the greatest propagation delay is now only one adder. Both data-flow diagrams output one sum every clock cycle.

FIGURE 10.a Slow FIR filter. A propagation delay of two adders is needed between clock cycles for this data-flow.

FIGURE 10.b Fast FIR filter. A propagation delay of only one adder is needed between clock cycles, yet this version has the same throughput as figure 10.a Both circuits are functionally identical.

2.5.2 Controller Synthesis

Controller synthesis consists of implementing a finite state machine from a state transition graph. The first step is logic *minimization* whereby the states are mapped to all possible distinct state assignments and the one with the fewest gates is chosen. For anything larger than a very small state machine, this is impractical. More effort has been placed into the second step, that of minimizing area by reducing the number of product terms.

The control finite state machine is usually implemented using one of the following architectures:

- A microprogrammed controller using Read Only Memory (ROM): where each bit of the microcode is mapped to select input on an FSM controlled unit such as a multiplexor or register enable. Each word of microcode is a state.
- A Finite State Machine with D flip-flops: where the logic is built from standard cells or logic arrays.

When the data path and controller synthesis are done, a *net list* will be generated for an optimized register-transfer level circuit with a finite state machine controller.

2.6 Logic Level Synthesis

At this level, the blocks of combinational logic are extracted from the register-transfer level description of the data-flow and controller and one of the following synthesis strategies are applied:

- Two-level logic minimization: Uses AND and OR gates as a basis for the sum of products, or products of sums. Espresso is a program developed at Berkeley which performs two-level logic minimization.
- Multi-level logic minimization: Involves factorizing out common terms from a sum of products, which usually increases the number of gates between the input and outputs. However, factorization can reduce the overall number of gates. This may result in simpler, smaller and possibly faster circuits. XOR gates can be used to implement symmetric sum of products such that the terms in each product are inverses of each other. Examples of commercial products which handle multi-level logic minimization are MIS II, SynopsysTM, SnergyTM and SmplicityTM.

Two level logic minimization takes advantage of redundant terms and outputs to minimize the sum of products. For example, the following expression for store_C generated from table 1:

$$\text{Store_C} = \bar{c} \bar{b} \bar{a} + \bar{c} \bar{b} a$$

can be reduced, using two level logic, to:

$$\text{Store_C} = \bar{c} \bar{b}$$

Multi-level logic minimization involves detecting groupings of identical logic and ensuring that only one instance of each group of identical logic is implemented. The outputs of each implementation will drive all logic gates which are dependant on the implemented groups. For example, given the two-level logic expression:

$$F = A\bar{B} + C\bar{B}$$

Using multi-level logic F can be reduced to:

$$F = \bar{B}(A + C)$$

Both minimization strategies achieve the same goal, that of reducing the surface area of the overall circuit.

2.7 Technology Mapping

This is not a level as described in the Y-chart in figure 3 on page 7, but rather as a process of implementing the structural description onto the physical domain at a specific level. This process can occur in several levels of abstraction. Three examples are mentioned.

- Algorithmic level: High level algorithms are mapped into an object code suitable for a specific micro-controlled processor.
- Register-transfer level: Hardware modules are mapped onto technology dependant macro cells. The macro cells consist of arithmetic units, registers, multiplexors, etc.
- Logic level: Logic gates are mapped onto technology dependant standard cells.

Once the mapping is complete, fan-out, propagation delays, and area usage can all be calculated for the technology chosen. A suitable clock frequency can then be determined and the circuit can be simulated, tested, and fabricated.

Chapter 3: A Silicon Compiler

The proceeding chapter provides an overview of behavioral synthesis and the many steps involved in the synthesis of a circuit. This chapter will focus on one specific set of steps which can be used to synthesis a behavioral description of a circuit.

3.1 The behavioral domain description in VHDL.

3.1.1 What is VHDL?

VHDL is a Hardware Description Language which was designed for the VHSIC program office of the United States department of defence in 1985 and was later standardized for the industry in 1987[7]. VHDL allows a circuit to be described in a behavioral like fashion, using language constructs like if statements, switch statements, and loops. VHDL allows the specification for an entire circuit to be described this way. The behavioral description can then be simulated to determine all necessary characteristics. The behavioral description can also be broken down into entities which can individually be simulated. This allows designers to develop each entity in parallel. Each stage of development can be tested with the simulation data from the other entities. Once the structural description of the entity has been developed, it can be substituted for the equivalent behavioral description. The input test data which was applied to the behavioral description can be used to verify the proper operation of the structural description. The original behavioral description can later be used for upgrades or even to develop a structural description better suited for a different discrete system, such as microprocessor based as opposed to synchronous logic circuits. VHDL however, can only be used to describe circuits which conform to a general stimulus-response model with discrete values.

3.1.2 VHDL behavioral descriptions:

A subset of the same behavioral description of a circuit can be used in circuit synthesis, thereby producing a structural description automatically. All descriptions in VHDL consist of the following three parts:

- **Components:** These represent blocks which can describe a part of a circuit at any level of abstraction.
- **Ports:** Each component has input, output, or input/output ports with which a component interfaces with the outside world. Global variables are not allowed.
- **Nets:** Components are connected to other components via nets. This is done in a hierarchical fashion where one super component will consist of many other sub components connected through their ports back to signals in the super component.

Each type of component is declared using an entity declaration, and defined using an accompanying architecture definition. The entity declaration describes all the ports of a component type, including their names and types. The architecture definition consists of the following:

- **Signal declarations:** The names and types of signals which are used within an architecture definition must be declared. Signals are similar to variables in a programming language except for a time delay which models the propagation delay in a circuit path. A signal only receives a change of value after the time delay.
- **Component declarations:** Any sub components which are a part of this super component type, must be declared.

The following algorithm, which was presented in section 2.4.1 on page 12, will be implemented in VHDL as an example.

```
c=0, p=1;
while (c < 2)
begin
    c = c + 1;
    p = p * c;
end
```

The VHDL entity declaration and architecture definition for the algorithm is shown below.

```
entity algorithm is
    port (Cin: in Integer; Pin: in Integer; Cout: out Integer; Pout: out Integer);
end algorithm;

architecture specification of algorithm is
    signal Mask: Boolean:= TRUE;           --<<SIGNAL DECLARATION
    signal C: Integer:=0;
    signal P: Integer:=1;
    signal Left, Right, Product: Integer:= 0;
```

```

component multiplier (L: in Integer;      --<<COMPONENT
                    R: in Integer;      --<<DECLARATION
                    Prod: out Integer);
end component;

```

The architecture body actually describes the behavior of the entity and may contain any of the following:

- **Component instantiation statements:** A component instantiation will create an instance of a declared component, link up its ports to declared signals using nets, and execute the component repeatedly.
- **Concurrent statements:** These statements will be repeatedly executed in parallel. One possible type of statement could be a Boolean expression assigned to a signal.
- **Process statements:** A process statement consists of a block of statements which are executed sequentially. The block as a whole is executed repeatedly.
- **Configuration specifications:** A configuration specification cross-references component declarations with their component types using the name of the entity declaration. Configuration specifications can also be declared outside the entity / architecture component description.

All the statement types described above are executed concurrently with respect to each other. Within a process statement, two classes of sequential control constructs are available:

- **Conditional control:** This includes the if then construct, if then else construct, if then elsif construct, and the case construct. They are very similar to what one would find in any programming language.
- **Iterative control:** There are three choices available, the simple loop construct with a conditional exit statement, the for loop, and the while loop.

The architecture body of algorithm is presented as follows. Note that a structurally defined multiplier has been used to demonstrate the use of a component instantiation. Otherwise a behavioral operation like $p = p * c$ would have been sufficient. Figure 11 on page 27 describes the operation of the VHDL example.

```

begin
    A_multiplier_instantiation: multiplier --<<A COMPONENT
        port map (Left, Right, Product);  --<<INSTANTIATION

    Left <= (P and not Mask) or (Left and Mask);--<<CONCURRENT

```

```

Right <= (C and not Mask) or (Right and Mask);--<<STATEMENTS

process newInput (Cin, Pin)  --<<A PROCESS activated by a new Cin or Pin
begin
    Mask <= TRUE;           --<< deactivate multiplier
    C <= Cin;                --<< load C
    P <= Pin;                --<< load P
    Mask <= FALSE;          --<<multiply C and P the first time.
end process

process calculate (Product)  --<<A PROCESS activated by
begin                        --<< a new Product
    if C<2                  --<<A CONDITIONAL CONTROL
    then
        Mask <= TRUE;      --<< deactivate multiplier
        C <= C + 1;         --<<update C
        P <= Product;       --<<update P
        Mask <= FALSE;     --<<multiply C and P
    else
        Cout <= C;          --<<output C
        Pout <= Product;    --<<output Product
    end if;
end process;
end specification;

configuration spec of counter is  --<<A CONFIGURATION SPECIFICATION
for specification
    for A_multiplier_instantiation: multiplier use
        entity MULTIPLIER(specification);  --<<declared externally
    end for;
end for;
end spec;

```

A test bench to instantiate the above VHDL program as well as the entity/architecture module to declare and define what MULTIPLIER is have not been included in the code. Figure 12 on page 28 shows how an example of how a complete VHDL program might look graphically. The test bench instantiates component 1, which in turn instantiates component 2 twice.

FIGURE 11. A graphical view of the VHDL counter example. Inputs are located at the top of statements and outputs are located at the bottom. No instantiations have yet occurred. An instantiated view of a fictitious circuit is shown in figure 12 on page 28.

As the main focus of this thesis concerns behavioral constructs, the syntax of the sequential control constructs mentioned above will be described very briefly. The underlined constructs have been interfaced with the bdsII control constructs described in chapter 4

if...then...end if: If the expression before *then* evaluates to TRUE, then the statement between *then* and *end if* is executed.

if...then...else...end if: Same as before, however if the expression before *then* evaluates to FALSE, then the statement between *else* and *end if* is executed

if...then...elsif...then.....end if: Same as before, except that *else* is replaced with *elsif*, the start of a new if statement. This construct can be recursively repeated until it is terminated with one of the two earlier constructs described above.

Test bench

Entity: ()

Architecture:

Component 1:

Entity:

Architecture:

Component 2:

Entity:

Architecture:

Instantiated Test bench (automatic):

FIGURE 12. A typical framework of a VHDL program. The components are defined on the left and instantiated on the right. Nets are used to connect the instantiated components together through their ports.

case...is when...end case: The expression between *case* and *is* determines which statement after the *when* in a list of *when*'s will be executed. There must be a *when* clause for all switch possibilities. The statement after the final *when* is terminated with *end case*. The case statement must be exhaustive and therefore no defaults are permitted¹.

loop...exit when.....end loop: All the statements between *loop* and *end loop* will be executed sequentially until the expression after *when* evaluates to true. The *exit when* clause may be placed as the first statement in the loop to simulate a while loop; it may be placed as the last statement in the loop to simulate a repeat until loop; or it may be placed anywhere in between.

for...in...to...loop.....end loop: All the statements between *loop* and *end loop* will be executed sequentially until the variable between *for* and *in* has incremented from the start value between *in* and *to* until the stop value between *to* and *loop*.

while...loop.....end loop: This loop is identical to the simple loop with the *exit* clause placed at the beginning. The expression in this version is located between *while* and *loop*.

Representing a problem specification as a behavioral description in VHDL can be considered a system-level synthesis. A complete example of a behavioral description in VHDL can be found in chapter 7. The next step, high-level synthesis, involves parsing the behavioral description.

3.2 The In-Core VHDL parser.

To perform high-level synthesis, each process of each component type must be parsed and converted into an internal representation. The nets between component instantiations and all concurrent processes must be stored internally as well.

1. While defaults are not permitted in VHDL, the generated CDFG switch control construct will always have a default to handle invalid values sent into the switch's fork node. Such invalid values will only occur in the event of a hardware glitch.

A VHDL compiler from the In-Core system is used to perform the front-end compilation[8]. A lexical analyzer and parser are used to process the VHDL source file into a temporary memory structure. This is later converted into an internal In-Core representation. In-Core, which was written by Spiros Beaucurois for his company called Functionality, was only capable of representing structural constructs. The internal representation has since been upgraded to allow behavioral constructs.

The In-Core representation stores an entity declaration as one general module and an architecture definition as another general module. One module can point to an instance of another to form a hierarchy of modules. This allows VHDL component instantiation statements to be stored hierarchically. Modules are only capable of representing structural components, therefore concurrent and process statements are stored as null module instances. Ports and signals are stored under both entity modules and architecture modules.

The algorithmic description of the circuit is encapsulated into the process statements of each architecture.

A combined data and control graph is used to represent the VHDL behavioral statements.

3.3 The bdsII internal representation.

A combined data and control-flow graph was chosen for the internal representation to effectively handle control dominated designs. Such designs are commonly found in ASIC's. Chapter 5 contains an example of a typical ASIC.

The bdsII internal data structure by Nortel is used for the internal representation. It consists of a network of combined data and control-flow graphs called CDFG's. A VHDL process statement is represented internally as a hierarchical tree structure where each vertex is a self contained CDFG. The process CDFG is initially generated. It will contain a node for each statement in a VHDL process. Data edges between the nodes will describe the flow of data from one statement to another. A separate control-flow edge will enter and

exit each node in sequence, clearly indicating the order of execution. Each of the nodes will themselves represent an instance of a CDFG graph which graphically describes the details of the statements. A control construct such as a condition or iterative construct as described above, will be represented graphically using fork and join nodes. These nodes as well as others will be described in detail in chapter 4.

A process CDFG, and sub CDFG's are generated for each VHDL process by invoking CDFG toolkit calls embedded in the methods of the ICvhdIParser class. The generated process CDFG is then hooked up to an implicit if-wait construct. All if-wait constructs will then be collected under an implicit loop construct. A root CDFG which will hold the entire circuit will be passed on to a front-end graphing program for visual presentation. The graphical interface will be described in more detail in chapter 6.

3.4 The Scheduler (Force directed)

Once the CDFG graphs for each process have been generated, the individual nodes which represent sequential statements must be scheduled. The CDFG graphs generated by the bdsII toolkits were designed to be scheduled using a scheduling algorithm called force directed scheduling[9][10]. While this time constrained scheduling algorithm was never implemented, it will be briefly described. Before this algorithm can be applied directly, two preliminary schedules must be performed to set the upper and lower control step bounds within which operations may be placed:

- As Soon As Possible scheduling (ASAP): Operations are scheduled immediately one after the other starting from the beginning.
- As Late As Possible scheduling (ALAP): Operations are scheduled in reverse immediately one before the other starting from the end.

Figure 13 contains an example of each type of schedule.

Note that the critical path in both schedules is the same, and is based entirely on data dependencies. This critical path determines the minimum number of control steps required, independent of the number of resources. This will be used as the time constraint.

FIGURE 13. As Soon As Possible scheduling versus As Late As Possible scheduling. The operational nodes have been numbered for identification.

Both schedules will require considerable resources for a few control steps and few resources for the remaining control steps. The goal is to generate a schedule between the ASAP and the ALAP which makes optimal use of resources. This can be done by carefully reassigning operations which are not a part of the critical path, to different control steps. Such operations are said to be mobile. The first addition operation in figure 13 is clearly mobile, while the second and third addition and unary minus operations are part of the critical path. For each type of operation, a histogram can be generated where the x axis represents the control steps, and the y axis represents the probability of an operation assigned to that control step based on both schedules. The histograms taken together are called the distribution graph for a CDFG. Figure 14 contains a distribution graph of the data-flow diagrams from figure 13.

To achieve the stated goal, each histogram in the distribution graph must be leveled out as much as possible. When an operation is finally assigned to a particular control step, it will

FIGURE 14.a The unbalanced distribution graph of probability verses control step, separated by operation.

FIGURE 14.b The balanced distribution graph of probability verses control step, separated by operation.

have a probability of one for that control step. All operations should be assigned one at a time such that the updated distribution graph becomes more levelled or balanced, if possible. For the example shown in figure 14.a, this can be accomplished by permanently assigning the first addition operation to control step 2, then all the “1”s in the figure will shift into the third column as shown in figure 14.b. Now it is possible for each of the operations to be implemented with the same functional unit. The indicator by which the assignment of an operation to a control step is measured is called force. The equations used to calculate the force for each operation is beyond the scope of this thesis.

3.5 Allocating and assigning functional units

The assignment of operations to functional units during scheduling is performed in a certain order. In the example of force directed scheduling, the following procedure is used for operational assignment.

1. Operations in the critical path are fixed to specific control steps (non-mobile), so they are assigned to functional units. This is shown in table 2 using the example in figure 13 on page 32. Initially, one functional unit is made available for each operation node.

TABLE 2. Critical path operation to functional unit assignment

control	FUNCTIONAL UNIT S			
step	adder1	adder2	adder3	subtractor1
1	operation 2			
2			operation 3	
3	operation 4			

2. Mobile operations which can only be placed in one of a few possible control steps (less mobile) are then assigned to functional units. These operations will be placed such that the distribution graph stays as balanced as possible. This is demonstrated in table 3 which is built upon the entries in table 2.

TABLE 3. Mobile operation to functional unit assignment.

control	FUNCTIONAL UNITS			
step	adder1	adder2	adder3	subtractor1
1	operation 2			
2	operation 1		operation 3	
3	operation 4			

3. Mobile operations which can be placed in one of many possible control steps (highly mobile) will then be assigned to functional units. In addition to ensuring that the distribution graph stays as balanced as possible, partially used functional units will be used as much as possible before allocating new ones. This last step is unnecessary for the example in figure 13.

From the results of table 3, it is obvious that the balanced data-flow graph can be implemented with only one adder and one subtracter. These two functional units can not be replaced with an ALU since both an adder and a subtracter is required simultaneously in control step 2.

The cross reference list in figure 15 on page 35 shows how functional units which are only assigned in some control steps, can be used in others, instead of allocating more functional units. For example, an ALU which is not in used for one control step can be used as an add operation, instead of allocating a new adder.

FIGURE 15. An operation to function unit cross-reference table for operation assignment.

3.6 The Register-Transfer level

At this level, the functional units must be replaced with hardware modules. For each functional unit, a corresponding hardware module must be selected from a library. The Selection is based on desired speed verses area constraints. For example, an abstract adder unit, can be replaced with a ripple-carry adder, a carry-look-ahead adder, or a carry-select adder. SynopsysTM is an example of one program which can select hardware units from functional units.

The remaining synthesis levels such as logic gates and standard cells are outside the topic of this thesis and will not be discussed.

Chapter 4: The Control/Data-Flow Graph (CDFG)

The primary goal of the CDFG is to graphically represent the flow of data from the initial input values to the final output values. This is accomplished using two flows, a data-flow which consists of data edges and data operation nodes, and a control-flow consists of control edges and control nodes. The structural part of the graph includes everything to do with the data-flow, while the behavioral part encompasses all parts of the control-flow. Both flows use the same building blocks, called elements, in construction.

4.1 The general architecture

The three basic elements which are used to construct a CDFG are nodes, terminals, and edges. Specialized nodes called *hiernodes* and port nodes allow data and control to be passed between CDFG's.

4.1.1 Nodes

The two basic type of nodes, data and control nodes, accept inputs and produce outputs. Each data-flow can have one input and output. A control-flow can also have one input, but may branch into other types of control-flows to produce multiple outputs. This is necessary for branching.

4.1.2 Terminals

Data or control enters and exits a node through data or control terminals on its circumference. Input terminals always appear at the top of a node and output terminals are always positioned at the bottom of a node. The terminals act as an interface between the node and its edges. Input terminals may be connected to at most one edge. Output terminals, usually data terminals, may be connected to multiple edges since a value can be read by multiple operations. Terminals may not be bidirectional.

4.1.3 Edges

Both data and control edges connect directly to terminals of the same type only. Each edge has exactly two ends which must connect to two distinct terminals. The flow of data or control along an edge is terminated by the distinct terminals. Therefore one terminal must be an input, and the other, an output.

An example of each of the basic elements described is identified in figure 16 on page 38.

FIGURE 16. An example of a basic control/data-flow graph.

4.1.4 Hiernodes

Hiernodes are special nodes which represent an entire CDFG within a CDFG. Since a CDFG has both a data-flow and a control-flow, the hiernode will have both data terminals. and control terminals. The only restriction is that there be only one input control terminal. Hiernodes are also used to represent behavioral constructs which are described below.

4.1.5 Port nodes

For each terminal in a hiernode, there exists a corresponding port node in the CDFG represented by the hiernode. All port nodes have an input and output terminal. The port nodes, which deliver data or control to the CDFG from the hiernode, are called input nodes. The input terminal of such a node is implicitly connected to the same terminal on the hiernode. An edge is not used. The output terminal of the port node connects to the rest of the CDFG using an edge.

Port nodes which return data or control back to the hiernode are called outputport nodes. The input terminal accepts data or control from the CDFG through an edge. The output terminal is then implicitly connected back to the same terminal on the hiernode. Once again this is done without using an edge.

An example of a hiernode, and the CDFG it represents is shown in figure 17 on page 39. The hiernode has been isolated from the rest of the CDFG it was connected to.

FIGURE 17. The framework of a hiernode. The external edges represent bindings between the terminals of the hiernode and its port nodes. The thick edges represent control-flow.

4.2 The data-flow

Before scheduling, it is assumed that each operation node will read its input values from a variable and then writes its output to a new variable. This implies that every edge in the data-flow represents a unique variable. However, if during scheduling, two or more dependent operations are somehow scheduled into the same control step, the data being passed between them will not be saved. In this instance, the data edge will not represent a latched variable but instead only the flow of a value. Operation nodes contain only data terminals. All operation nodes are implicitly bound to a control edge in the order that they appear in

the graph. There are two graphing styles used to graphically describe the reading and writing of values to variables. Both are described briefly.

4.2.1 Read write node style

The edges connected to a data operation node are terminated at their opposite ends with a read or write node, which represents a variable being accessed or modified respectively. The edge will represent the flow of the value itself. Edges which are flowing into a data operation will be initiated with a read node while the edges flowing out of a data operation will be terminated with a write node. If the value which has just been written to a variable, is needed immediately for another data operation, then a corresponding read node will immediately follow the write node. An edge will carry the value from the read node to the next data operation. There is no direct connection between the write and read nodes used for different operations, as a result the overall graph is disconnected. Every tree in the forest will contain one operation. Figure 18 on page 40 contains an example of a read write node assignment style.

FIGURE 18. An example of the read write node style. The resulting value from the addition is stored in the variable C. The value is then read from variable C. This is functionally identical to the pure data-flow style, described below. The appearance however, is more complicated.

4.2.2 Pure data-flow style

The edges in this style represent the variable itself, not the value. The value is written to or read from the variable at the terminal interface between the edge and the data node. Consequently there are no read write nodes. The output terminal of one operation node is connected directly to the input terminal of another operation node with a single data edge. For simplicity and ease of understanding, only the pure data-flow style of the bdsII internal data structure is used. Consequently this is the data-flow style used in all CDFG figures in this thesis, with the exception of figure 18.

4.3 The control-flow

Nodes which contain control terminals are part of the control-flow of a CDFG. These include hiernodes, which have been described already, and control-flow nodes.

Control-flow nodes allow the control-flow to branch to alternate pathways. Since all data operations are bound to some control-flow, then selecting an alternate pathway causes alternate operations to be executed. The hiernodes in a control-flow represent a block of data-flow and the control edges between the hiernodes describe the flow-of-control between the blocks. Behavioral constructs can be built up allowing many different groups of data operations to be executed depending on a condition. The basic building block for these constructs is the fork node and the join node.

4.3.1 The fork node

This node consists of a single input data terminal, a single input control terminal, and multiple output control terminals. The value on the data input will be used as a switch to determine which output control terminal will be connected to the input. There will be exactly one connection from the input control terminal to exactly one output control terminal at any time. Figure on page 42 contains two examples of fork nodes.

FIGURE 19. The fork node. The data input selects the output control terminal for the input control terminal. A value of 0 selects the left most control output of the fork node, followed by incrementing values to select the next output to the right.

4.3.2 The join node

The join node works the opposite way around. There are many input control terminals and only one output control terminal. There is however no formal switch for the join node at this level of detail. If the control-flow for the CDFG was constructed properly, then exactly one and only one of the input control terminals to the join node will be active at any one time. Therefore a switch is not expressly needed and so all the input control terminals are connected to the single output control terminal all of the time. If a data-flow is associated with any of the control-flows, then an input data terminal is added for each input control-flow to ensure that the corresponding output data terminal of the join node has an equivalent input data terminal for each input control-flow. For example, the join node in figure 20 on page 42 has two input control-flows. Originally the left control-flow had only a dark

FIGURE 20. A two input flow join node. Only one of the input flows may be active at any one time.

blue data-flow associated with it and the right control-flow had a light blue data-flow associated with it. For a proper join node however, both control-flows must have both data-flows associated with each of them similar to the way two fractions cannot be added until both of their denominators contain the same factors. For those control-flows which do not have the same associated data-flow, the missing data-flows will be retrieved from the respective inport node of the CDFG as shown in the join node of figure 21 on page 45. In the event that there is no inport node for the needed data-flow, an inport data node will automatically be generated to provide the data-flow. The generated inport data node will automatically be associated with the input normal control node because any other type of control-flow is not allowed for inport nodes. The input terminals will be ordered such that each data-flow will be grouped with its associated control-flow.

4.3.3 Types of control-flows

Control-flows are classified into five types. Normal, break, continue, return and goto flows. They all function identically, but are treated differently in behavioral constructs. Any CDFG is only allowed to have one output control node for each type. This in turn limits the number of output control terminals on a corresponding hiernode to five. Each type of output control terminal can also have its own respective output data terminals. This permits each control-flow to assign its own unique value to a common variable.

Normal flow: This flow represents the main flow, all other control-flows should eventually join with this flow.

Break flow: A break flow represents a flow which should be joined with the normal flow at the end of a behavioral construct.

Continue flow: A flow which should be joined with the normal flow at the beginning of a behavioral construct.

Return flow: A flow which should only be joined with the normal flow at the bottom of a CDFG.

Goto flow: This flow has not yet been implemented into any practical behavioral constructs.

These control-flows are used together with the control-flow nodes to build three main types of behavioral constructs. It is the design and construction of these behavioral constructs which represents a major portion of the thesis work.

4.3.4 The if construct

This construct consists of 5 nodes. Of these, the condition, left, and right hiernode must be constructed in advance with the restrictions described below. The left and right nodes are optional.

- **Condition hiernode:** This node has one input and one output normal control terminal, one or more input data terminals, and a single output data terminal which is used as a switch in a fork node.
- **Fork node:** The output terminals from the condition hiernode are connected directly to input terminals on the fork node through edges. There are exactly two output normal control terminals, a left and a right. The input data is interpreted as a switch where a value of 0 selects the left terminal and a value of 1 selects the right terminal.
- **Left hiernode:** The left output control terminal from the fork node will be connected directly to a input control terminal using an edge. The output control-flow may be of any type described above and multiple output control terminals may be used, up to one of each type.
- **Right hiernode:** The functionality of the right hiernode is the same as that of the left hiernode except that the right output control terminal from the fork node will be connected directly to a input control terminal using an edge.
- **Join node:** The join node will join dangling normal control-flows only. This means that unless both the left and right output normal control-flows propagate through the left and right hiernodes, then a normal join node will not be necessary. If either the left or right hiernodes are absent then, the corresponding output normal control-flow from the fork node will propagate down to the join node if the join node is needed.

Any other types of control-flows which might originate from the left or right hiernode will be left dangling. These flows will be handled by the CDFG wrap up procedure.

Figure on page 45 contains an example of a then only if construct.

FIGURE 21. A generic if construct without an else. Note: The fork outputs are reversed in the mode and the nodes are all given the same shapes.

Figure 22 on page 46 demonstrates a more complicated if construct.

FIGURE 22. A generic if construct with all control-flow types except for goto. Note: The fork outputs are reversed in the model and the nodes are all given the same shapes.

4.3.5 The switch construct

This construct is somewhat more complicated than the previous. It consists of 6 nodes where nodes 3 and 4 are repeated for each additional switch case.

- Condition hiernode: This is identical to the condition hiernode described in the if con-

struct with the following exception. The single output data terminal will provide many possible values from 0 to n.

- Fork node: The properties of this node are similar to the previously described fork node, however the left and right output control terminals are replaced with an output control terminal corresponding to each switch case.
- Case hiernode: There is one case hiernode for each switch case, however the corresponding output control terminal of the fork node is only directly connected the input control terminal of the case hiernode for the first switch case. The second, third, fourth, etc... case hiernodes are separated from their respective output control terminals by a normal join node if applicable. In most situations, each case hiernode will have an output break control terminal and possibly one or more output data terminals related to the break control terminal.
- Normal Join node: The objective of the join node is to connect the output normal control-flow from the previous case hiernode with the next output normal control terminal from the fork node. The output normal control terminal of the join node then connects directly to the next case hiernode. If the previous case hiernode has no output normal control terminal, then this join node will not be inserted.
- Final case hiernode: The input normal control terminal of this node will be connected to the output normal control node of the previous join node, if applicable. A output break control terminal is redundant for this node since output normal control terminal will be connected to a final join node directly.
- Break join node: All the dangling break control-flows and their associated data-flows will be merged together in one large join node. The output control terminal will be normal. The associated data-flows will appear as output data terminals, one for each variable, and they will be associated with the output normal control terminal.
- Final normal join node: The output normal control-flow from the final case hiernode, and the break join node, and associated data-flows, will be merged in this join node. At the end of the construct there will be only one normal control-flow.

Continue, return, and goto control-flows which may have originated from case hiernodes will be ignored by the switch construct. These flows will be handled by the CDFG wrap up procedure.

Figure 23 on page 48 contains an example of a switch construct.

FIGURE 23. A generic switch construct with normal and break control-flow. Note: The fork outputs are reversed in the model and the nodes are all given the same shapes.

4.3.6 The loop construct

The generic loop construct consists of seven nodes. Of these, the last two may be unnecessary. Any type of loop can be derived from these basic nodes.

- The loop join node: This node will join a left and a right control-flow. The right input terminals will be connected directly to the inport nodes, from which initial values for all data-flows will be obtained as well as the normal control-flow. The left input terminals are a part of the loop back and as such will be connected to edges originating from output terminals within the next four nodes described.
- Top hiernode: This hiernode should contain operations which are to be executed at least once, even if the loop exit condition evaluates to true.
- Condition hiernode: This is identical to the condition hiernode described in the if construct.
- Fork node: This node is also identical to the fork node described in the if construct. The left output control terminal will be connected directly to the bottom hiernode such that as long as the condition evaluates to false, the loop will continue. The right output control terminal will be connected directly or indirectly to an outport normal control node.
- Bottom hiernode: This hiernode should contain operations which are only to be executed if the condition evaluates to false for each iteration.
- Continue join node(s): There are three possible scenarios: If an output continue control-flow terminal is present in the top or bottom hiernode, then the continue control-flow along with any associated data-flow is merged with the normal control-flow from the bottom hiernode in a join node. The output terminals are connected directly to the left input terminals of the loop join. If both the top and bottom hiernode have a continue control-flow, then the two control-flows and their associated data-flows are merged together in a separate join node. The output terminals of that node are then fed in the continue join. If on the third hand, there are no continue flows, then the output terminals of the bottom hiernode are connected directly to the left side of the loop join node.
- Break join node(s): The three same scenarios are possible as described above. Break control-flows from the top or bottom hiernodes are handled the same way as continue flows. The break flow, if any, will be merged with the right fork output terminals of the fork node. The output control-flow from the break join node will become the new normal control-flow for the CDFG, including any associated data-flows.

An example of a loop construct can be seen in Figure 24 on page 50.

In some popular loop constructs, the top hiernode and bottom hiernode are not used simultaneously. Three types of loops are described without considering continue or break flows.

- While loop: This type of loop is only executed if the condition evaluates to true. The

FIGURE 24. A generic loop construct with top and bottom body simultaneously. Note: The fork outputs are reversed in the model and the nodes are all given the same shapes.

structure of such a loop in a CDFG would involve the following nodes: loop join, condition, fork, and bottom. The condition hiernode itself however must evaluate to false to loop again and evaluate to true to exit.

- For loop: This loop is actually a sub set of the while loop and as such contains the same structure. The bottom hiernode increments a variable until it causes the condition hiern-

ode itself to evaluate to true.

- Repeat until loop: This loop is executed at least once before the condition is evaluated. The loop continues as long as the condition evaluates to false. The structure needed to implement this type of loop in a CDFG uses the following nodes: loop join, top, condition, and fork. The left side of the fork will immediately loop back to the loop join while the right side of the fork exits the loop normally. The condition hiernode must evaluate to false to loop again and true to exit.

4.3.7 Wrapping up a CDFG

The behavioral constructs described above do not generate output nodes for the CDFG, this task is performed by a wrap up routine. This routine performs the following steps.

1. Break, continue, return, and goto control-flows which have dangling edges in the CDFG are each separately wrapped up. If there are two or more control-flows of the same type, then a join node is generated to merge the flows as well as any associated data-flow into one control-flow of that type. Since only one normal control-flow is allowed after a behavioral construct has been completed, a normal control join node will never be necessary.
2. An output control node is generated for each control-flow type and their input terminals are connected to the appropriate dangling control-flow edge.
3. An output data node is generated for each associated data-flow and their input terminals are connected in a similar fashion. The outputs are grouped in order of their associated control-flows. If the variable associated with a data-flow does not exist in the super CDFG where the hiernode resides, then it must be a local variable and therefore an output data node will not be created since it will have no meaning in the superCDFG.

Figure 22 on page 46 contains an example of a wrapped up CDFG.

4.3.8 CDFG Hookup

The process of hooking up a wrapped up CDFG consists of creating a hiernode for that CDFG where its terminals will be automatically connected to all inports and outputs in the CDFG, and then inserting that hiernode into the superCDFG. The hiernode will be

inserted by connecting the dangling normal control-flow in the super CDFG to the control in terminal on the hiernode. All other in terminal connections then flow. The normal control out terminal then becomes the new normal control-flow for the super CDFG. Control and data dependencies are then generated for all the out terminals of the hiernode so that their dangling edges can be connected to another hiernode, or they can be terminated with output nodes when the super CDFG is wrapped up.

4.4 Displaying CDFG's

Once a CDFG has been created, it should be displayed so that the synthesis tool designers can verify its correctness. The process by which a CDFG can be displayed is by no means standardized. When CDFG's are automatically generated, the placement of their nodes and routes by which edges are drawn between nodes must be carefully determined by routing and placement algorithms. Otherwise the displayed CDFG's may be so difficult to understand as to defeat the purpose of trying to verify their correctness. Determining exactly how to properly draw a CDFG is not a trivial problem.

4.4.1 The postscript generation utility

The only mechanism available for displaying CDFG before this thesis was started was by generating a postscript file of a CDFG and then viewing the file with a postscript viewer. Figure 25 on page 53 contains an example of a CDFG drawn in this manner. Data dependent nodes are drawn one under another while independent nodes are drawn adjacently. Three basic flaws with this postscript file as described as follows:

1. No attempt was made to draw terminals. Therefore all edges sharing the same source and destination nodes were drawn over each other. This causes key details such as the then and else control-flows from the fork node to be hidden. Imagine how a switch construct such as the one shown in figure 36 on page 67 would appear.
2. The layout of the nodes in figure 25 causes edges to be drawn through them, detracting from the overall appearance. This occurs with the cond, fork and then nodes.

FIGURE 25. A postscript drawing of a CDFG. Black edges represent data-flow while grey edges represent control-flow.

3. The size of the page restricts the number of nodes which can be drawn underneath each other or sideways. CDFG's such as the one shown in figure 36 on page 67 would either only be partially drawn or nodes would overlap each other.

For the purposes of this thesis, a much more sophisticated system was necessary, one which could handle many nodes, while trying to keep edges apart such that they could clearly be distinguished.

4.4.2 A crossover reduction drawing algorithm with unique edge colors.

The problem of distinguishing edges in a displayed CDFG was approached using three strategies.

1. Drawing evenly spaced terminals on the nodes to prevent edges from converging together at their source or destination nodes.
2. Color coding data-flow edges by variable so their values can easily be followed from start to finish.
3. Counting the number of crossed edges and crossed nodes when drawing each and every edge and then choosing the path between two nodes with the fewest number of crossovers.

These strategies were implemented when redrawing the CDFG from figure 25 on page 53. The redrawn CDFG is shown in figure 21 on page 45. The same strategies were used when drawing other CDFG's shown in figure 22 on page 46, figure 23 on page 48, and figure 24 on page 50. Based on these results, it was decided rerouting edges was not sufficient in reducing the number of crossovers. A new strategy was needed to uncross multiple edges between common source and destination nodes.

4.4.3 Reducing crossovers through improved ordering of nodes and terminals

The crossovers between the normal control-flow inport shown in figure 24 on page 50, for example, and the A, B, and C data-flow inport nodes can be eliminated by simply reordering all the inport nodes such that the control-flow inport node is always to the right of the data-flow inport nodes. This strategy was found to be very successful in general. Changes in the ordering of inport nodes tended to propagate all the way down to the outport nodes. Furthermore, a well ordered CDFG had a tendency to produce well ordered terminals on

hiernodes representing the CDFG. The following general characteristics were found to reduce edge crossovers:

- Always keep control-flow inports and outputs to the right of data-flow inports and outports. This will ensure that control-flow terminals on hiernodes are also to the right. This works especially well for loops as shown in figure 30 on page 63.
- Stagger adjacent hiernodes such that common data-flows needed by two or more hiernodes will not immediately cross through all but the last hiernode. An example of staggered hiernodes is shown in figure 43 on page 78.

The following characteristics were found to make CDFGs more readable in general:

- In the event that crossovers are inevitable, right angled crossovers or approximations are easiest to follow. Figure 36 on page 67 is a good example of this.
- Edges should contain a minimum number of bends. Bends which cannot be easily eliminated should be placed such that the final destination of the edge can be estimated from the trajectory of its source. An oversimplified conceptual example is shown in figure 26.

FIGURE 26. To bend or not to bend. Clearly the left diagram is more complicated to follow than the right. This is because the trajectory of the edges in the left diagram lead to the destinations of the edges.

- Figure 26 also demonstrates that allowing certain avoidable crossovers may improve the readability of a CDFG.

4.4.4 An algorithm for node placement during CDFG drawing.

The order in which nodes were drawn was found to be very significant. The best results were obtained by first determining the longest control-flow path from the inport to output nodes. All the nodes along this path were then drawn one underneath another since control dependencies existed between them. The remaining nodes were then drawn starting with the output nodes. This was done to ensure that each edge connected to a common output terminal will be drawn, since the unique input terminals for each edge will be drawn first.

Figure 27 on page 56 demonstrates what happens if the nodes along the longest control-flow path are not drawn first. The correctly drawn CDFG can be found in figure 40 on

FIGURE 27. A carelessly drawn CDFG. The corrected version can be found in figure 40 on page 71

page 71.

Chapter 5: A real example: “The traffic light”

The control/data-flow graph described in the previous chapter is generated using two levels of toolkit functions. A description of how to use these functions is described in appendix E. These functions can be invoked directly to generate a CDFG description of a circuit. However, as appendix E demonstrates, this is a very awkward and time-consuming chore. This provides the motivation to develop an interface to a well known circuit design language. The interface itself will invoke the toolkit functions, thus shielding the designer from specific details about the toolkit functions.

Rather than attempt to describe the one-to-one mapping between each VHDL statement and the toolkit functions, a simple example of a VHDL circuit design will be presented along with figures from a generated CDFG hierarchy. The graphical pictures not only serve as a verification tool to ensure that the generated control constructs are correct, but will also show the interactions between circuit components, including unexpected feedback loops which might otherwise be missed.

The traffic light example is intended to demonstrate how well the VHDL behavioral constructs are converted into control constructs embedded within the CDFGs. As such, the dominant flow in the example is the control-flow. There are therefore many branches and few data operations. This is not a concern as the primary interest is the control-flow. Data-flow can easily be added, however this would complicate the example needlessly.

5.1 The specification

The traffic light circuit[14] at the system level, will have one input and two outputs. The input indicates the presence of a car on a farm road either trying to cross a highway, or turning left onto the highway. Both the highway and the farm road have a set of three lights to direct the traffic. This constitutes a standard traffic light intersection, with the following exception, the light on the highway direction always stays green unless the input is enabled, meaning a car is waiting on the farm road.

Each output represents the color of the signal for one road. One for the highway and the other for the farm road. Each output can have three values, red, yellow, or green. Two timers are needed to control the cycling of the traffic lights. A long timer which determines the duration of the green signal for the highway, and the duration of the green signal for the farm road. A short timer defines the duration of the yellow signal for each road type. The duration of the red signal for each road type is determined by adding that of the long and short timers. There are two exceptions with respect to timing which are noted as follows:

- The green signal for the highway may persist until a car is present on the farm road. For this exception, the long timer only serves to define a minimum duration.
- The green signal for the farm road may expire immediately if the car has left the farm road. For this exception, the long timer only serves to define a maximum duration.

5.2 System level synthesis

The problem described above must be manually broken down into processes so that high level synthesis can be applied to each process. The traffic light circuit can be broken down into the following five processes.

- The controller: This process determines which state the circuit will be in at any given moment.
- The highway traffic signal: This process will determine the correct value of the highway traffic signal dependant on the current state of the controller.
- The farm road traffic signal: This process will determine the correct value of the farm road traffic signal, also dependant on the current state of the controller.
- Long timer: A long resetable counter with an output which signals a time-out.
- Short timer: A short resetable counter with an output which signals a time-out.

Figure 28 on page 59 shows the signals which are needed to allow the five processes to interact with each other. Each process will be discussed individually.

5.2.1 The controller

The finite state machine for the controller is given in figure 29 on page 60. The labels in regular font are inputs, those in bold italics are outputs. The state itself is also an output as

FIGURE 28. The top level diagram showing the signal flow between the five processes.

shown in figure 28. Note that there are no red-light states. As shown in section 5.2.2 and section 5.2.3, the red light signals are derived from the green and yellow states of the opposite traffic light crossing.

5.2.2 The highway traffic signal

This process simply determines the value of the highway traffic signal, depending on the state of the controller. For each state listed below, the desired value is given.

Highway Light Green: Highway traffic signal = Green.
 Highway Light Yellow: Highway traffic signal = Yellow
 Farm road Light Green: Highway traffic signal = Red.
 Farm road Light Yellow: Highway traffic signal = Red.

FIGURE 29. The finite state machine for the controller process. The labels in regular font are inputs, those in bold italics are Meally outputs, and the states themselves in ovals are Moore outputs. Red light states are unnecessary as the red lights are derived from the green and yellow states of the opposite traffic light crossing

5.2.3 The farm road traffic signal

This process simply determines the value of the farm road traffic signal, depending on the state of the controller. For each state listed below, the desired value is given.

Highway Light Green: Farm road traffic signal = Red.

Highway Light Yellow: Farm road traffic signal = Red

Farm road Light Green: Farm road traffic signal = Green.

Farm road Light Yellow: Farm road traffic signal = Yellow.

5.2.4 The short timer.

This process accepts a start signal, and then outputs a timed-out-short signal after a certain time delay. A counter will be used to implement this timer. The start signal will reset the counter and the output. An adder will be used to increment the value up to a maximum count, which is given externally to the circuit, then the output will be set and will stay set until the counter is reset by the start signal.

5.2.5 The long timer

This process accepts a start signal, and then outputs a timed-out-long signal after a certain time delay. The functionality of this counter is identical to that of the short timer. The maximum count will be provided externally to the circuit.

5.3 High level synthesis

Now that the specification has been divided into well defined processes, each of these processes may be described algorithmically in VHDL. Initially a package called “Traffic Package” defines the values of the traffic signals, and the states of the controller process.

The description of the traffic light itself is divided into two parts, the first part is the entity which describes the external inputs and outputs of the circuit, the second part is the architecture which describes the algorithms and components within the circuit. The algorithms can be described as processes and/or concurrent statements.

The entity of this circuit will be named “traffic light controller” and its architecture specification will define the five processes described above. Two of these processes will each be implemented as a type of concurrent statement called a conditional signal assignment. The remaining processes will be implemented as VHDL processes due to their increased complexity.

The only other entity, called “TLC_Test”, will generate a test input and determine the duration of the long time and the short time. The test input and duration value will be fed

into a component instantiation with generic parameters and ports identical to those defined in the traffic light controller entity/architecture.

Finally, a configuration statement will link up the component instantiation with its entity declaration and architecture definition.

The complete VHDL code is listed as follows. Selected control/data-flow graphs representing portions of the code have been inserted for comparison. The blue edges represent individual data-flow while the pink edges represent control-flow.

-- Package for encapsulating standard types and constants¹

```
package Standard is
    type Boolean is (FALSE, TRUE);
    type Time is range -2**63 to 2**63-1
        units
            fs;          -- femtosecond
            ps = 1000 fs;-- picosecond
            ns = 1000 ps;-- nanosecond
            us = 1000 ns;-- microsecond
            ms = 1000 us;-- millisecond
            sec = 1000 ms;-- second
            min = 60 sec;-- minute
            hr = 60 min;-- hour
        end units;
end Standard;
```

-- Package for encapsulating types and constants for the Traffic Light Controller²

```
package Traffic_Package is
    type Color is (Green, Yellow, Red, Unknown);
    -- Type and Subtype Declarations
    type State is (Highway_Light_Green, Highway_Light_Yellow,
        Farmroad_Light_Green, Farmroad_Light_Yellow);
end Traffic_Package;
```

-
1. The types defined in this standard package are used by all VHDL programs and therefore are not specific to the traffic light controller example. The package is made available to the traffic_light_controller entity/architecture using the VHDL statement: `use work.Standard.all;`
 2. The types in this package will be made available for use throughout the traffic_light_controller entity/architecture with the VHDL statement: `use work.Traffic_Package.all.`

-- Entity declaration defining the inputs and outputs for the traffic light controller

```

use work.Standard.all;
use work.Traffic_Package.all;
entity traffic_light_controller is

```

The values for generic declarations are defined when the entity/architecture is instantiated as shown on page 72.

```

    generic (Long_Time: Time; -- Minimum/Maximum green light duration
            Short_Time: Time -- yellow light duration
    );
    port (Car_On_Farmroad: in Boolean;
          Highway_Light: out Color;
          Farmroad_Light: out Color
    );
end traffic_light_controller;

```

-- Architectural body for the specification of the traffic light controller

```

architecture specification of traffic_light_controller is
    --signal for holding the current state of the system
    signal Present_State: State:= Highway_Light_Green;
    --signal for implementing the timing mechanism
    signal Timed_Out_Long, Timed_Out_Short, Start_Timer:
        Boolean:= FALSE

```

The signal declarations can be seen in figure 30 on page 63.

begin -- SPECIFICATION

-- PROCESS STATEMENT WHICH IMPLEMENTS THE STATE MACHINE

Controller_Process:

process begin

case Present_State is

when Highway_Light_Green =>

if Car_On_Farmroad and Timed_Out_Long then

The condition in the above if statement is shown in figure 31 on page 64

Start_Timer <= not Start_Timer;

Present_State <= Highway_Light_Yellow;

end if;

when Highway_Light_Yellow =>

The switch label Highway_Light_Yellow is shown in figure 32 on page 65.

if Timed_Out_Short then

Start_Timer <= not Start_Timer

The assignment in the previous line is shown in figure 33 on page 65;

Present_State <= Farmroad_Light_Green;

The direct assignment from the previous line is shown in figure 34 on page 66

end if;

when Farmroad_Light_Green =>

if not Car_On_Farmroad or Timed_Out_Long then

Start_Timer <= not Start_Timer;

Present_State <= Farmroad_Light_Yellow;

end if;

when Farmroad_Light_Yellow =>

FIGURE 32. Another condition CDFG. This compares the switch select `Present_State` to the switch label `Highway_light_Yellow`. The switch and relevant case as shown as follows:

case `Present_State` is

 when `Highway_Light_Yellow` =>

FIGURE 33. An unary operation CDFG. The assignment is shown as follows:
`Start_Timer <= not Start_Timer;`

if `Timed_Out_Short` then
 `Start_Timer <= not Start_Timer;`
 `Present_State <= Highway_Light_Green;`
end if;

The above if statement is shown in figure 35 on page 66;

FIGURE 34. A direct assignment CDFG.
Present_State <= Farmroad_Light_Green;

FIGURE 35. A complete if statement CDFG.The control-flow divides into an else and then branch which are rejoined with a join node. The VHDL if statement is shown below:
 if Timed_Out_Short then
 Start_Timer <= not Start_Timer;
 Present_State <= Highway_Light_Green;
 end if;

end case;

The complete switch statement is shown in figure 36 on page 67.

wait on Car_On_Farmroad, Timed_Out_Long, Timed_Out_Short;
The above wait statement is shown in figure 37 on page 68
 end process;

```

-- Conditional signal assignment to set highway light
Highway_Traffic_Signal:
  with Present_State select
    Highway_Light <= Green when Highway_Light_Green,
                      Yellow when Highway_Light_Yellow,
                      Red when Farmroad_Light_Green I

```

Farmroad_Light_Yellow;

The complete conditional concurrent statement is shown in figure 38 on page 69

-- CONDITIONAL SIGNAL STATEMENT TO SET FARMROAD LIGHT

Farmroad_Traffic_Signal:

with Present_State select

Farmroad_Light <= Green when Farmroad_Light_Green,

Yellow when Farmroad_Light_Yellow,

Red when Highway_Light_Green I

Highway_Light_Yellow;

-- PROCESS STATEMENT TO IMPLEMENT SHORT COUNTER

Short_Timer:

process

variable counter: Time:= 0 ns;

begin

Timed_Out_Short <= FALSE;

FIGURE 38. A CDFG of a conditional signal statements shown below. The word with in VHDL denotes a switch and the word when denotes individual cases of a switch construct:

with Present_State select

Highway_Light <= Green when Highway_Light_Green,
 Yellow when Highway_Light_Yellow,
 Red when Farmroad_Light_Green |
 Farmroad_Light_Yellow;

for counter in 0 ns to Short_Time loop

The condition, counter < Short_Time, embedded in the above for loop and the auto-increment mechanism, counter++, are shown in figure 39 on page 70.

end loop;

Timed_Out_Short <= TRUE;

```

        wait on Start_Timer;
    end process;

```

-- PROCESS STATEMENT TO IMPLEMENT LONG COUNTER

Long_Timer:

```

process
    variable counter: Time:= 0 ns;
begin
    Timed_Out_Long <= FALSE;
    for counter in 0 ns to Long_Time loop
    end loop;

```

The above loop as a whole is shown in part a of figure 40 on page 71

```

        Timed_Out_Long <= TRUE;

```

The for loop and two signal assignments on either side are shown in part b of figure 40 on page 71.

```

        wait on Start_Timer;
    end process;
end specification;

```

All the architecture statements in the above architecture body is shown in figure 43 on page 78.

FIGURE 40.a. The CDFG of a complete for loop. The hiernode bot_body contains the auto-incrementing mechanism

FIGURE 40.b. This CDFG represents the process labelled. Long_Timer.

The CDFG's in figures 40.a and 40.b, were generated from the following VHDL code. Since the variable Counter is defined outside the loop statement but altered within the loop statement, the new value must be returned by the loop as shown in figure 40.a, even if it is not used as shown in figure 40.b

```
variable counter: Time:= 0 ns;
begin
    Timed_Out_Long <= FALSE;
    for counter in 0 ns to Long_Time loop
    end loop;
    Timed_Out_Long <= TRUE;
```

The following Entity/Architecture called TLC_Test is a test bench and therefore cannot be synthesized. Its only purpose is to instantiate a component of traffic_light_controller

-- ENTITY DECLARATION FOR TEST BENCH: NO INPUTS OR OUTPUTS

entity TLC_Test is end TLC_Test;

--ARCHITECTURAL BODY FOR TEST BENCH FOR THE TRAFFIC LIGHT CONTROLLER

```
use work.Standard.all;
use work.Traffic_Package.all;
architecture Test of TLC_Test is
    --SIGNAL FOR STIMULATING TRAFFIC LIGHT CONTROLLER
    signal Car_On_Farm_Road: Boolean:= FALSE;
    --SIGNALS FOR CATCHING OUTPUT OF TRAFFIC LIGHT Controller
    signal Highway: Color:= Green;
    signal Farm_road: Color:= Red;
    --COMPONENT TEMPLATE FOR INSTANTIATING A
    --TRAFFIC LIGHT CONTROLLER
    component TLC
        generic (Long_Time: Time;
                Short_Time: Time);
        port (Car_On_Farmroad: in Boolean;
              Highway_Light: out Color;
              Farmroad_Light: out Color);
    end component;
begin
    --INSTANTIATE THE TRAFFIC LIGHT CONTROLLER
    Controller: TLC
        generic map (Long_Time => 5 ns, Short_Time => 2 ns)
        port map (Car_On_Farmroad => Car_On_Farm_Road,
                  Highway_Light => Highway,
                  Farmroad_Light => Farm_road);
The above Instantiation is shown in figure 41 on page 73.
    --STIMULATE THE TRAFFIC LIGHT CONTROLLER INPUT
    Car_On_Farm_Road <= FALSE,
        TRUE after 1 ns,
        FALSE after 3 ns,
        TRUE after 10 ns,
        FALSE after 20 ns;

end Test;
```

--Configuration specification for TLC_Test entity

*This configuration statement will link the component instantiation called **Controller: TLC** with its entity declaration and architecture definition called **traffic_light_controller**.*

```

use work.all;
configuration spec of TLC_Test is
  for Test
    for Controller: TLC use
      entity work.traffic_light_controller(specification);
    end for;
  end for;
end spec;

```

5.4 The XbdsII executable.

The control/data-flow graphs embedded in the description of the entity/architecture called **traffic_light_controller** are snapshots of the XbdsII graphical user interface shown in figure 42 on page 74.

FIGURE 42. The graphical user interface for bdsII. The main window with the drawing area is shown in the background while the open file dialog box is shown in the foreground. the VHDL source file "trafficLight.vhdl" is about to be compiled.

5.4.1 Loading

A VHDL source file must first be selected by choosing the menu item *open* from the file menu. The dialog window in figure 42 will then appear with a default directory of `"./vhdl"` and a filter mask of `"*.vhd"`. This means that any files with an extension of `".vhd"`, which reside in a subdirectory called `"vhdl"` of the current directory, will be immediately visible in the file selection dialog box. The user may browse through file system directories to select the desired vhd source code file. The desired file may be loaded by either double clicking on its name, or by highlighting the file name and selecting the "Ok" button.

Each self contained circuit in the vhdl source file will be parsed into a CDFG hierarchy and become an entry in a design library. A self contained circuit is one which has no ports in its entity declaration (i.e. a test bench). Each library entry will appear as an unconnected node in the drawing area of the main window of the XbdsII executable.

5.4.2 Viewing CDFG's

The contents of any hiernode can be viewed simply by clicking once on the left mouse button while the mouse pointer is within the rectangular perimeter of the node. The rectangular perimeter is defined as the largest possible rectangle which can be drawn inside the shape of the node. The CDFG the hiernode resides in can be redrawn simply by clicking in the same fashion inside one of the inport or outport nodes of the hiernode's CDFG. In this way the entire CDFG hierarchy of a circuit description can be perused, one CDFG at a time.

5.4.3 Exiting

The executable can be terminated simply by selecting the menu item *exit* from the file menu. The graphical representation of the various CDFG's will be removed from memory, followed by the CDFGs themselves. Finally, the destruct functions for the toolkits will be called and the executable itself will terminate.

5.5 The Hierarchy Structure

This section will describe how all parts of the vhdl circuit description are displayed, starting from the top level library entry and focusing all the way down to sequential statements in a process. Currently the bdsII internal representation only models sequential control-flow. This implies that the control-flow will not spontaneously split into multiple control-flows unconditionally. Therefore CDFG's cannot be used to model concurrency between processes and concurrent statements in VHDL which implicitly execute concurrently and continuously. Nevertheless, graphs are generated anyway for all levels of the VHDL circuit description with the exception of time delayed signal assignment in a test bench.

Graphs which contain hiernodes of concurrent statements such as the CDFG in figure 37 on page 68, must simply not be considered real control/data-flow graphs until the model can be improved to represent concurrency.

5.5.1 The test bench.

When a library entry is selected in the design library, the test bench for that circuit will be displayed. This will consist only of hiernodes representing components which have been instantiated by the test bench. The terminals of the hiernode itself represent the external ports of the instantiated component as described by its entity declaration. The terminals will be connected to test bench signals which will be represented with inports and out-ports. The time dependent signal assignments generated by the test bench itself cannot be synthesized. Only the initial value of each test bench signal will be shown. An example of an instantiated component is shown in figure 41 on page 73.

5.5.2 The Implicit architectural loop

In VHDL, each process, concurrent statement, or component instantiation executes forever. There is therefore an implicit loop. For VHDL processes this loop is triggered by changes in signals specified by the wait statement in the body of a process. For concurrent statements or component instantiations, the implicit loop executes without pausing such that any changes occurring at its inputs are immediately propagated to its outputs.

While attempting to graph these implicit loops, it was decided that the graph would be far easier to follow if the VHDL implicit architectural loops were extrapolated into one large external loop as is shown in figure 30 on page 63. The node labeled "top body" represents all the processes, concurrent statements, and component instantiations defined within the architecture body of a VHDL entity declaration and architecture definition. The contents of the hiernode are shown in figure 43 on page 78.

The right input to the loop join from figure 30 represents the initial values of signals which have been defined in the architecture body. These signal are then passed through to the

architecture body represented by the top_body hiernode. The values of any signals which are modified are then passed out of the hiernode where they enter the right side of the loop join and can then be reused by the architecture body. As the loop demonstrates, all statements are executed forever. The output normal control-flow has no meaning and serves only to allow the graphing functions to operate correctly

The mechanism by which the VHDL processes in the architecture body are suspended until a signal change, without affecting the concurrent statements or component instantiations will be described in the next section.

The contents of the implicit loop, meaning the process itself, will be executed once for each signal change detected. A concurrent statement does not have any VHDL wait statement. It's implicit loop executes without suspending for any reason.

5.5.3 The Architecture Body

All the processes, concurrent statement, and component instantiations defined in an architecture body must operate concurrently and independently of each other, except for the signals that they share. The order of their appearance in figure 43 on page 78 is therefore meaningless. Each hiernode in this figure represents a process or concurrent statement. In this example, there are no component instantiations. All three types will be described.

Concurrent statements

These statements execute endlessly such that any signal changes are immediately propagated to the outputs without any intermediate latching. The switch statement in figure 38 on page 69 is an example of a concurrent statement.

Processes

Processes consists of a list of sequential statements which are only executed once for each trigger from a wait statement. The outputs must then be latched between triggers. Signal changes will not be propagated until the next trigger occurs. The switch statement in figure 36 on page 67 is an example of a sequential statement because it is only executed once

FIGURE 43. This diagram represents the variable dependencies between all the architecture statements in the entity/architecture labelled `traffic_light_controller`. Additionally interactions between the statements including possible unexpected feedback loops will also be shown. All the CDFGs represented by the hiernodes are executed concurrently with respect to each other. The order shown in the diagram is therefore meaningless.

every time the wait statement in figure 37 on page 68 evaluates to true. An if construct such as the one shown in figure 44 on page 79 is used as the mechanism to execute the sequential statements inside the then hiernode every time the wait statement in the condition hiernode evaluates to true. This is why all process statements in figure 43 on page 78 appear as “if” hiernodes. The contents of the then hiernode, as shown in figure 45 on page 79, contains a hiernode for each sequential statement in the process, of which there is only the switch sequential statement for this example.

Component instantiation statements

An instantiation creates an instance of a component so it can be hooked up to the outside world through its input and output ports defined in it’s entity declaration and architecture

FIGURE 44. The wait if construct. The process contained in the then hiernode will not be executed until the wait condition contained in the condition hiernode evaluates to true

FIGURE 45. This CDFG represents the contents of the then hiernode from figure 44 on page 79..

definition. Component instantiation allows the component to be reused within other components. The test bench in figure 41 on page 73 contains a component instantiation of the traffic light controller entity definition and architecture definition.

5.5.4 VHDL statement encapsulation

In an effort to maintain readability for larger circuit designs, each set of VHDL statements are encapsulated into one CDFG. For example, each set of statements to be executed depending on a control construct, will be represented by a hiernode. One example of such a hiernode is the then hiernode shown in figure 44 on page 79. The expanded CDFG for this hiernode will itself consist of a hiernode for each statement to be executed as shown in figure 45 on page 79. The expanded CDFG of a hiernode statement will then contain the actual statement as shown in figure 36 on page 67. The encapsulation will add a considerable number of levels to the CDFG hierarchy. This can be a disadvantage in that the reader can become easily disoriented.

5.6 Register transfer level synthesis

The control/data-flow graph hierarchy described above can be saved to a file which may be read in by another program to perform register transfer level synthesis. The file will contain the contents of the memory used to hold the structure of the CDFG hierarchy. BdsII libraries can be linked into the register transfer level synthesis program to allow it to access and understand the structures in this file. For example, one of these libraries is called the persistent memory module. Its job is to store and retrieve all the class instances which are subclasses of the root class "BDSRoot". This library is not used in this thesis.

Chapter 6: System Implementation

The system consists of three packages, a VHDL compiler called In-Core, bdsII itself, and a front-end graphical interface called XbdsII. These packages were initially developed separately on different Unix platforms, however they have all been recompiled together on a common platform. The first two packages are written in C++ while the C only source code for the latter is automatically generated. The three packages will be collectively known as bds97. Each package will be described in detail. Existing structures and functions will be explained, followed by additions made during the course of this thesis work.

6.1 Platform

The current platform being used is a Sun SparcStation IPX (sparc 2) with 48 megabytes of memory and a 1 gigabyte hard-drive. The current version of the operating system is Solaris 2.5. The Common Desktop Environment (CDE), version 1.0, is used to execute the front-end graphical interface for bdsII. Bds97 has been compiled with gnu C++, version 2.7.2

6.2 OverView

Figure 46 presents an overview of the system from the point of view of the user. A VHDL file must first be opened. The bdsII internal representation will then be generated, and the user can browse through the CDFG hierarchy interactively using the mouse. Finally, the application must be terminated and the memory containing the internal representation must be released and returned back to the operating system.

Figure 47 on page 83 presents another overview of the system, this time from the point of view of a programmer. The system is divided up into levels of abstraction, the highest of which is the user input event handler and the lowest being the XWindow system library function calls needed to draw the circles and lines onto the application window. The user will generate an event such as selecting a VHDL file to be parsed. This event will invoke a function at the graphical user interface level which will in turn invoke a function at the In-

Open event:

1. Executes a function in the c/c++ external interface file called "createRootCDFG" with a parameter "filename".
 - This function will invoke the In-Core parser and pass it the filename of the VHDL source file.
 - For each entity declaration or architecture definition, the parser will invoke bdsII toolkit functions to create a CDFGData.
 - Subsequently created CDFGData's will become super or subCDFG's of existing ones such that a tree or hierarchy of CDFGData will be formed.
 - The root CDFGData will be returned to the external interface function where it will be stored in a global variable.
2. Executes a function in the c/c++ external interface file called "initializeCDFGGraphics" and passes the drawing area widget from the graphical user interface.
 - This function will reserve shades of color blue, one pink color, and a purple background color in the system's color palette. A backplane for the widget, called a pixel map, will be created for drawing.
 - The method "createGraph" will be invoked on the root CDFGData.
 - This method will create an un-initialized Graph class for the root CDFGDATA and then do the same recursively for all subCDFG's.
 - The first Graph associated with the root CDFGData will be returned.
 - The method "initialize" will be invoked on the Graph and passed the drawing area widget, reserved colors, and pixel map.
 - This function will be recursively invoked for all sub Graphs, thereby transmitting the widget, colors, and pixel map to all Graphs.
 - Then all nodes in the associated CDFGData will be graphed by creating a Vertex for each node and assigning coordinates, and by creating an Arc for each edge between nodes and assigning a shade of color blue.
 - The method "reSize" will be invoked on the root Graph only and the scaling factor passed as a parameter.
 - The coordinates of all Vertices and Arcs will be multiplied by the scaling factor and drawn onto the pixel map.
 - The pixel map will then be painted onto the drawing area widget.

Left mouse button release while in the drawing area widget event:

1. Executes a function in the c/c++ external interface file called "CDFGGraphicsInput".
 - This function will call an input method in the current Graph which determines if the mouse coordinates at the time the left button was released, was within a hiernode.
 - If so, then the CDFG for that hiernode is retrieved and the Graph class for that CDFG is returned.
 - If the mouse coordinates were within an inport or outport node, then the superCDFG is returned.
2. If a new Graph is returned then the reSize method is invoked upon it with the current scalingFactor.
3. The rePaint method is invoked upon the new Graph to paint its pixel map into the drawing area widget.

Exit event:

1. Executes a function in the c/c++ external interface file called "removeCDFGGraphics".
 - This function will recursively destroy all the Graph classes including their pens Vertices and Arcs.
 - It will then release the color palette entries containing the reserved colors and finally destroy the pixel map.
2. Executes a function in the c/c++ external interface file called "destroyRootCDFG".
 - This function will destroy the entire bdsII internal representation in memory by recursively releasing the memory for each CDFG.
3. The graphical user interface terminates and the application window disappears off the screen.

FIGURE 46. User triggered events.The first event is triggered by selecting a filename from a dialog box. The second is triggered by clicking anywhere in the drawing area widget. The third event is triggered by selecting the exit menu item from the file menu. See figure S2 on page 94 for a more detailed view of the events.

FIGURE 47. Levels of abstraction from the GUI down to the Xlibrary functions. The boldface parts were designed and implemented during the course of this thesis while italicized parts were modified. A description of the four components of the bottom level can be found in appendix C.

Core parser level. As the parser compiles the VHDL code, it will repeatedly call functions at the bdsII toolkit level to generate CDFGs. Later on, once the entire CDFG hierarchy has been created, initialization and graphing function calls will propagate from the GUI level to the graphical elements level, where the graphical elements for each CDFG will be generated. Finally, the graphical elements of each CDFG will be drawn onto a backplane which will be referred to as a pixel map. The pixel map for one CDFG is then painted onto the drawing area inside the application window. The above discussion can be more clearly seen in Figure 48 on page 84 which demonstrates how each of the three packages relate to each other. The contents of each package will be described in the remainder of this chapter starting with the In-Core parser. Figure 52 on page 94 at the back of this chapter provides a very detailed view of the calling sequence described above which is initiated by the user as shown in figure 46 on page 82.

FIGURE 48. A diagram of the three packages which together constitute bds97. The triangles represent series of cooperating tasks. The ovals define the information which is passed between packages. The cylinder at the bottom of the diagram represents the storage in memory of the entire data structure for bds97. The boldface and shaded sections were designed and implemented during the course of this thesis while italicized sections represent modifications.

6.3 In-Core Classes

This program consists of three groups of C++ classes: VAST classes, the ICvhdl Parser class, and the ICbase classes. In addition, a textual parser is used to initially read the VHDL source file. Each group will be described separately.

6.3.1 The parser

The VHDL textual source code to be compiled is initially broken down into tokens using a *lexical analyzer*. The tokens are then grouped together into rules with a parser program. The analyzer program is generated automatically from an input file of regular expressions using a GNU lexical analyzer generator called Flex. The parser program is generated automatically from a grammar input file using a GNU textual parser called Bison. Both programs produce c++ code which can then be compiled into an executable.

The tokens consist of *syntactic groupings* of characters. Some of these groupings will be keywords in VHDL, in which case they can be replaced with a semantic value. Other groupings will consist of number values, variable names, literal strings, etc...The parser program will attempt to match up a sequence of tokens to a grammar rule. Each rule contains an action which is executed once a rule has been found to fit a sequence. An example of a sequence is an if...then...else statement. The action consists of C or C++ code to store the sequence as a structure or class. If the parser were a C or C++ compiler, then the action for an if...then...else sequence would be to generate the appropriate machine code to actually execute the branch of the if statement.

6.3.2 The VAST classes

Each sequence of tokens matched to a VHDL statement is stored as an instance of a c++ class. There is a unique class for every different kind of VHDL statement or clause. All of these classes are collectively known as VAST classes. Their purpose is only to store the VHDL source code in temporary, reusable memory in case the bison parser program aborts due to a syntax error. Should such an event occur, the memory allocated to the

VAST class instances will not be freed before the parser restarts. Instead the same block of memory will be reused to avoid eventually running out of memory altogether. Since the memory is temporary and reusable, the VAST class instances must themselves be parsed and stored in a more permanent part of memory as ICvhdlParser class instances. This should only occur when the parser program has parsed a complete block of syntax error free VHDL code, such as an entity declaration or architecture definition.

6.3.3 The ICvhdlParser class

A VHDL file is parsed by first creating an instance of an ICvhdlParser class and then invoking a method contained in the instance, with the name of the VHDL file passed as a parameter. This method then invokes the parser program. Once a complete block of VHDL code has been parsed, the parser in turn calls a method back in the ICvhdlParser class called yypostParse. This method will generate a module, and add it to a library of modules which is contained within the instance of ICvhdlParser. This step actually generates the entire internal representation of the VHDL source code, module by module. Parsing then continues until all VHDL blocks have been parsed, then the parser program exits and control returns back to the ICvhdlParser method which also exits back to its caller.

At this stage the entire VHDL textual file has been converted into a memory structure which consists of modules with ports, nets, terminals, and instances which can relate to other modules. This entire design can be access by the program which created the instance of ICvhdlParser through get methods.

6.3.4 The ICbase classes

The entire design, which is contained within an instance of ICvhdlParser, as described in section 6.3.3, is actually an instance of the topmost class in a hierarchy of classes grouped under the name ICbase classes. A diagram containing an example of instantiated ICbase classes is shown in figure 49 on page 87. There is only once instance of the topmost class, which is called ICdesign. An instance of an ICdesign will consist of a list of ICLibrary instances, one for each VHDL source file parsed. Each instance of an ICLibrary class con-

FIGURE 49. A model of the memory image of the In-Core base class structure. Each library represents a set of related entity/architectures. A module in one library may instantiate a component defined by an entity/architecture in another as shown. An ICconnector is used to provide an interface with which to link up the component instantiation to its entity/architecture. The link itself consists of a set of ICbitnet called an ICnet. All the libraries are collected under an ICDesign.

tains a list of ICModule instances. The two main types of instances of a ICModule class are ENTITY and ARCHITECTURE. The other types of instances are reserved for gener-

ating internal representations from other hardware languages such as verilog and will not be discussed here. Each ICMODULE class instance contains each of the following attributes:

- ICparameter instances: This class is used to represent a VHDL generic parameter. An example of a VHDL generic statement can be found in the component instantiation in figure 41 on page 73.
- ICport instances: This class describes a port of a VHDL entity declaration.
- ICnet instances: An ICnet class represents a VHDL signal declaration.
- ICinstance instances: An ICinstance represents one statement in a VHDL architecture body.

The last attribute, ICinstance, has no meaning for an ICMODULE instance of type ENTITY and would therefore be set to null. The ICinstance class, can have several different types of instances, some of them are described as follows:

MODULE_NAME: Not bound to anything, undefined.

PRIMITIVE: Implemented at a lower level.

MODULE_DEF: Implemented by another module;

BEHAVIOUR: Implemented by a CDFGData.

COMPONENT: An instantiation of another module.

All types of ICinstance instances have a moduleName. If the ICinstance instance has a corresponding ICMODULE instance, then the moduleName will be identical to the ICMODULE instance name.

Connection Classes

A set of ICbase classes are dedicated to making connections between an ICMODULE's ICport instances and instances of ICinstance owned by the ICMODULE. The word instance will be assumed implicitly hereafter when referring to ICbase class names for reasons of simplicity.

The topmost connection class is called an `ICInterface`. It is an abstract class of `ICPort` and `ICTerminal`. The functionality of an `ICTerminal` class is identical to that of an `ICPort`, except that `ICTerminal` belongs with `ICInstance`, while `ICPort` belongs with `ICModule`. An attribute in each of these classes determines the direction of data-flow. Both connector classes are made up of `ICConnection`, one for each bit in an `ICPort` or `ICTerminal`. Each `ICConnection` in an `ICPort` is connected to the corresponding `ICConnection` in an `ICTerminal` using an `ICBitnet`. The set of `ICBitnet` class instances are grouped under the `ICBase` class, `ICNet`.

6.3.5 Type classes

In addition to the above described classes, a set of five abstract base type classes are provided from which language dependant types can be derived. The basic `ICBase` types are described as follows.

- `ICBasicDataType`: Used for single types, such as integer, boolean, enumerated, etc...
- `ICRecordDataType`: Used for deriving structures.
- `ICUnconArrayDataType`: Used for deriving arrays with indexes from 0 to infinity, or up to the limit of the integer type used for indexing.
- `ICFixedArrayDataType`: Used to derive fixed length arrays with indexes starting from 0.
- `ICConstrArrayDataType`: Used to derive arrays with a specific range of indexes such as a range from 9 to 15. `ICFixedArrayDataType` should be used instead to save overhead if the range will be starting at zero.

For the purposes of this thesis, two VHDL types have been implemented, an `ICVhdlEnumDataType` class and an `ICVhdlPhysicalType` class. Both classes were derived from the `ICBasicDataType` as shown in figure 50.

Expression data is used to represent specific values of a data type. The class `ICExprData` is provided as a base from which language specific expression data classes can be derived.

FIGURE 50. Hierarchy of In-Core data type classes. The classes defined under ICbase data types may be used to derive language specific data types. ICvhdlPhysicaltype and ICvhdlEnumDataType are two such examples. The shading indicates that they were added during the course of this thesis.

The following contains some of the expression data classes that have been implemented for VHDL.

- ICvhdlInteger: Expressions involving integers should be derived from this class
- ICvhdlReal: Expressions involving real numbers should be derived from this class
- ICvhdlChar: Expressions involving characters should be derived from this class
- ICvhdlString: Expressions involving strings should be derived from this class
- ICvhdlBinary: Expressions involving binary values should be derived from this class
- ICvhdlVar0Ref: Values stored in an instance of this class are implicitly of type ICvhdlEnumDataType.
- ICvhdlPhysConstant: Values stored in an instance of this class are implicitly of type ICvhdlPhysicalType. Values such as 10 μ s, 4 mV, 10 nF, etc... can be stored as an ICvhdlPhysConstant. The units will automatically be understood and handled appropriately.

Figure 51 contains a hierarchy of expression data classes starting with ICexprData as the root. Most of these classes are not yet in use as a data type has not been created for them. The exceptions are ICvhdlPhysConstant and ICvhdlVarORef.

FIGURE 51. The Hierarchy of In-Core expression data classes. Each of the classes at the leaves must belong to a data type class from figure 50 on page 90 if they are to be used.

6.3.6 Limitations

The In-Core ICbase class structure is capable of representing fixed links only. This includes component connections to their ICmodules and assignment of values to typed variables. VHDL behavioral constructs, such as a concurrent signal assignment or a selected signal assignment, involves making links to typed variables which are switchable depending on a condition such as if C=1 then F=A+B else F = A-B. This ability cannot be supported with fixed links. Therefore the ICbase classes are only used to represent a VHDL structural description.

Additional methods have been added the ICvhdlParser class for the purposes of this thesis. The additional methods invoke bdsII toolkit calls which in turn, construct CDFGs which represent the VHDL behavioral descriptions. Appendix E provides examples on how these toolkits are used to create CDFGs. All the CDFG's will be connected in a hierarchy of CDFG nodes, the root of which can be referenced from an instance variable in an ICmodule architecture instance.

6.4 Implementation details

For a discussion on the implementation details of bds97, please refer to appendix D.

6.5 Overview

Figure 52 shows the calling sequence for all the components of bds97 from the instant the application program is started until it is shut down. Cascaded methods actually represent recursion.

Chapter 7: Conclusions

7.1 Summary

The master objective is to build a synthesis system. Specifically, this will be performed by creating a netlist of a behavioral VHDL program. Before a netlist can be generated, the behavior of the program must be described using an internal representation. This has been achieved during the course of this thesis through a set of objectives which also includes a graphical browser for viewing the behavioral internal representation. Each objective and the tasks performed to meet that objective will be stated as follows:

1. The behavioral parts of the bdsII internal representation were missing and had to be implemented. This objective was met as described:
 - Additional functions were added to the bdsII construct toolkit for the creation of switch and loop control constructs. The function responsible for generating an if control construct was rewritten. Code responsible for handling alternate control-flow such as break, continue and return were *encapsulated* into self contained functions which were then invoked from the construct functions. Supporting functions responsible for generating elements such as inports, hiernode terminals, and outports, were redesigned to determine the best ordering of these elements to reduce the probability of crossed edges during graphing.
2. A behavioral compiler was needed to parse a well known and understood hardware language, called VHDL. An existing structural VDHL compiler was upgraded as follows:
 - The compiler was upgraded to handle behavioral statements by implementing two new data types, handling process and concurrent statements, recognizing sequential assignment and control statements within a process, and separating expressions into sequences of one or two input operations.
3. The compiler had to be interfaced with bdsII to generate an internal representation from the behavioral VHDL program. The interface was implemented as described:
 - A set of class methods were added to the compiler to generate CDFG's from the VHDL behavioral statements. The methods invoked functions from two different bdsII toolkits. Those in the lower level CDFG toolkit allowed sub CDFG's to be created, such as data-flow comparisons and data-flow assignments. Those in the higher-level construct toolkit used the data-flow CDFG's as blocks to create control constructs.

4. A mechanism for showing the internal representation to the outside world was deemed necessary to test and verify the correct operation of the parser and bdsII toolkit functions which generated the internal representation. To this end, the following tasks were performed:

- A set of graphing classes were developed to visualize a CDFG by traversing its nodes twice. The first pass draws all nodes along the longest path from inport to outport node to provide a point of reference for the remaining nodes. Adjacent hiernodes are automatically staggered to reduce the probability of edges from one hiernode crossing through another. The second pass draws the remaining nodes from outport to inport. This implies that edges will be drawn from their input terminals to their output terminals to ensure that multiple edges belonging to common output terminals will not be missed.
- A front-end graphical user interface was developed to display the graphed CDFGs onto a screen through an application window. The user interface, which was implemented in the Common Development Environment, allows the user to browse in and out of CDFGs by selecting hiernodes or port nodes respectively using a mouse.

7.2 Innovations

This thesis has primarily been a major task in integrating multiple software packages, each of which required modifications and additions to make it possible for them to communicate with each other. The bdsII construct toolkit in particular was incomplete and required extensive work before it could be used by the In-Core VHDL parser. Most of this work required foresight in the sense of implementing a consistent model for the underlying representation for multiple control-flow types such as break, continue and return flow, when developing the if, switch, and loop control constructs. This model is flexible enough to allow each kind of control construct to handle multiple control-flow types the same way such that code is reused.

Due to the complexity of the resulting CDFGs, a sophisticated display system was designed and implemented to display the CDFGs. This graphical browser is far superior to the embedded postscript utility in that the internal representation can be viewed as a hierarchy of CDFGs, one level at a time. A subCDFG can be displayed by simply clicking with the mouse on any of its hiernodes. The superCDFG can be displayed by clicking on

any inport or outport nodes along the circumference of a CDFG. Color has been added and the placements of nodes re-ordered in an effort to make each CDFG more readable. The browser itself is plug and play in the sense that all the CDFGs can be viewed for any VHDL file without recompiling any code. In fact the XbdsII application can be executed, used, and understood by anyone with only a rudimentary knowledge of Unix systems.

7.3 Benefits

The XbdsII program implemented during the course of this thesis will provide a synthesis system designer with an intermediate view of their synthesized circuit. This will allow the designer to spot errors and inefficiencies quite easily, and correct for them.

Synthesis system designers have traditionally verified the correctness of their CDFGs by laboriously hand drawing the CDFGs from lists of pointer references produced by their hardware description language to CDFG compilers. This process was only manageable for very simple CDFGs in the order of 20 nodes or less. The verification of complicated behavioral constructs with embedded hiernodes would have been far too tedious. By providing the synthesis systems designers with an automatic view of their synthesized circuit in the form of a CDFG graphical browser, CDFGs can be verified at a glance.

The common development environment was selected as the underlying graphical user interface platform because of its popularity on a number of different unix based computer systems such as Hewlet Packard and Sun. Furthermore, the CDE represents a new standard for the X window system, virtually ensuring that future window platforms will be backward compatible with the CDE. It is hoped that the program will be upgraded to allow the graphical browser to include scheduling, resource allocation, and resource assignment information.

7.4 Future improvements

This thesis was undertaken with the expectation that this would be an ongoing project of which I have not started, nor will I be finishing. Several improvements are needed on vari-

ous aspects of the program of which I had no time to implement myself. I will mention some of them here as well as suggestions concerning future features for the benefit of my successor.

- The control/data-flow graph model needs to be improved to permit concurrency between processes.
- The robustness of the graphical user interface could be improved by “idiot proofing” the widgets to prevent core dumps should they be activated before a CDFG hierarchy is successfully loaded.
- Error detection and reporting of logical errors in the VHDL source code or signals which are read before being written does not yet exist. The In-Core VHDL parser can only detect syntax errors.
- More can be done to reduce the number of crossovers in graphs. This can be achieved by a better placement of nodes and by rerouting edges around nodes once they have been placed.
- Improvements need to be made to the wrap up function to restrict the creation of outport data nodes for dependency variables to those which are actually needed in the super CDFG. Currently outport nodes are created for all dependency variables in a CDFG so that their values can be accessed by the corresponding out terminals on the hiernode. If the CDFG where the hiernode resides does not need to read values from all of the out terminals, then the unneeded ones will be left unconnected. This problem can be seen in figure 40 on page 71. The solution lies in the ability to detect when an outport node will not be needed and then avoid creating it in the first place.
- All outport data nodes are bound to a control-flow which indicates when a value will be available at the outport node. In special cases however, data must be transmitted to the out terminals of the hiernode before the execution of the hiernode itself is complete. This special case must be handled when generating a register-transfer level structure. The output *Timed_Out_Short* in figure 40 on page 71 is an example of such a case.
- The persistent memory module should be tested and a memory dump to a file attempted.
- Only a subset of all the VHDL behavioral statements actually generate control/data-flow graphs, the remaining statements should also be implemented in the In-Core parser.
- At present, the In-Core parser only instantiates an entity declaration and architecture definition the first time. Multiple instantiations should be implemented.

The following suggestions represent a considerable future programming effort.

- A tracking system which visually shows where the current CDFG being displayed is in relation with the rest of the CDFG hierarchy would allow the user to keep track of

where they are.

- Prior to generating a register-transfer level structure for each CDFG, all the embedded data-flows must be scheduled. Currently genetic algorithms appear to yield good optimizations.

Appendix A: Related Work

The bdsII control/data-flow graph was derived from an exploratory project of a high level synthesis prototype system called HAL. A complete history of bdsII can be found in Appendix A. The prototype was designed and implemented by Pierre G. Paulin as part of his Ph.D research[18]. The original CDFG only had two control constructs, the if construct and the loop construct. There were no concepts of hiernodes, therefore simple data-flow graphs were used instead of hiernodes. Consequently, the entire circuit was described in one CDFG. Multiple control-flow types such as break control-flows, return control-flows etc... were not possible. The mechanics of the data-flow and the data operation are however were very similar.

7.5 A data-flow only model

A high-level synthesis system by Kazutoshi Wakabayashi uses tree-structured Control-Flow Graphs (tCFG) to represent the hardware description, and CDFGs for internal representation[11]. The data-flow is very similar to the data-flow used in bdsII, however there is no actual control-flow. Fork node and join nodes are controlled by control edges from comparator operator nodes directly. There is also no actual control constructs, only fork and join nodes.

7.6 Sequencing Graphs

Another high-level synthesis system which specializes in control dominated ASIC designs, uses a sequencing graph called Sequencing Intermediate Form (SIF) as the internal representation[12]. The graph consists of simple and complex vertices. Simple vertices consist of computational operations, read/write operations, input/output ports or message passing commands. The latter allows synchronization of parallel processes. This is not directly supported in bdsII. Complex vertices are basically specialized hiernodes of which there are three types: function calls, conditionals, and loops. Both the function call vertex and loop vertex consist of a sequencing graph. The sequencing graph of the function vertex is executed once and then returns, that of the loop vertex however, is executed from top

to bottom repeatedly until an exit condition is satisfied. There are no loop back edges allowed in any sequencing graphs, they must be acyclic. The loop vertex consists of many distinct sequencing graphs, each representing a branch of the conditional vertex. Only one branch will be executed.

The regular structure of this model makes pure data-flows very difficult to draw. Instead a read write style as described in 4.2.1 is used. Control edges from complex vertices do not connect directly to operations, read vertices are inserted in between. The output edges of the read vertices are data-flows which enter into the operation vertex. The output data-flow from the operation vertex then enters a write vertex. The control-flow then resumes. This is demonstrated in figure 53.

The control-flow in a sequencing graph can split multiple ways simultaneously to support multiple execution threads. This is also demonstrated in figure 53 when B and C are both read simultaneously. Multiple execution threads can be implemented in a future version of bdsII by binding two or more independent operation nodes to parallel control-flows. The timing constraints are more detailed in sequencing graphs.

7.7 Value Trace (VT)

Another high-level synthesis program called the system architect's workbench uses a combined data/control-flow graph[13]. It has been named a value trace graph, presumably because the data-flow is visible at all times, unlike sequencing graphs. The value trace graph uses a pure data-flow style. The control nodes consist of select operators for implementing if and switch constructs, call operators which encapsulate a value trace graph within a value trace graph, and basic blocks. There is no control operator however to implement loops. A basic block contains a data-flow graph of operations. The data between basic blocks is connected via data-flow edges. These characteristics make the value trace graph model very similar to the bdsII model.

FIGURE 53. A sequencing graph demonstrating a conditional hiernode and a multiple execution thread.

Appendix B: The history of BDS

The concept of BDS (Bnr, Dds, Sil) was no doubt at least partially inspired by previously published papers in the field, as is usually the case with new innovative concepts. In one such paper, the idea of extracting timing constraints from a behavioral description is presented [15]. The use of timing constraints would allow a behavioral description to be synthesized rather than simply simulated as has been previously done. The timing constraints could be represented as a directed graph where the vertices represent operations and the edges define the timing constraints. Event times could then be assigned to the vertices such that the timing constraints on the directed edges would be preserved.

In a BNR report [16] written by Pierre Paulin, Cliff Liem, Trevor May, and John Knight in April 1992, paired data-flow graphs (DFG) and a control-flow and timing graphs (CFG) were used to represent behavioral descriptions. The first consisted only of pure data-flow while the later was used to represent conditional branching using fork and join vertices. The CFG was limited to one input control edge and one output control edge. Simple bindings were used to link up corresponding components in both types of graphs. A separate DFG/ CFG made it possible for a behavioral construct to be interpreted in more ways than one, from a synthesis system point of view. An early version of this work was originally described in Pierre G. Paulin's Ph.D. thesis[18].

In October, 1992, a simple language compiler (VOP to BDS) was added which to rapidly create BDS graphs [17]. The VOP compiler was rewritten in November and December by Trevor May and Francis Langlois. The updated version parsed a more sophisticated input language which was named ocuhdl (O Carleton University Hardware Description Language). The following year, as work proceeded, it became obvious that the basic DFG/ CDG was becoming limited in the behavioral constructs it could represent.

By May, 1994, Trevor May, Christopher Donawa, and Francis Langlois wrote a specification for a redesigned BDS based on a merged DFG and CFG which was called a Control/

Data-Flow Graph (CDFG). The concept of a CDFG was originally described in Emil Girczyc's Ph.D. thesis[20].

Besides many other improvements, multiple output control-flows such as a return flow, break flow, and continue flow were added. The redesigned BDS was given the name BDSII. Chris Donawa and Francis Langlois worked on BDSII until December 1994.

At that time BNR cancelled the project. During the first quarter of 1995, Francis Langlois ported the source code, as is, to a GNU C++ compiler at Carleton University so that a graduate student could continue implementing BDSII.

The following people have also made contributions to the project. Yatish Kumar, Arthur Castongany, and Shailesh Sutarwala.

Appendix C: The X Window system.

The X Window system[19] was designed to provide programmers with a hardware and software device independent way of manipulating graphics on a computer screen through the use of windows. The system is divided into layers. The most primitive layer accessible to programmers is called the Xlib, short for XLibrary. Other layers are built on top of this base layer. Each additional layer allows the programmer to manipulate graphics using high-level functions. Up to this day, all the layers consist of c library routines which can be invoked by a programmer with the help of special include files to define the structures expected in the parameter lists. The layers are listed below followed by a description of each.

- **Xlib:** This layer contains functions for creating bitmaps, color palettes, drawing tools, etc...
- **X Toolkit Intrinsics:** This layer uses the functions from Xlib to create a set of higher-level functions which allow self-contained user-interfaces to be created, such as scroll-bars, buttons, etc...
- **Widget Sets:** This layer provides the programmer with a set of pre-defined user-interfaces, called widgets, to choose from. Although the programmer will determine the color and size of a widget, the look and feel of the widget will be determined by the widget set.
- **X Window Manager:** The background pop-up menus, window attributes, and basic application utilities, which are packaged with an X window manager, usually consist of widgets from a specific widget set. This does not prevent applications based on a different widget sets from being executed under the X window manager.

C.1 Xlib

Because all of the other layers mentioned are based on Xlib, it is possible to write an entire X window application using only XLib functions. This would provide the programmer with a maximum level of flexibility, however, at the same time, require a considerable coding effort. Common subroutines, such as creating a window, dialog boxes, menus, etc... can be more efficiently handled by a widget set. One aspect of graphics, which cannot be wrapped up into an easy to use widget, is that of drawing graphs or pictures onto a screen. High-level widgets are available to create a drawing area in an application window, com-

plete with automatic scroll bars. However one must still resort to simple drawing functions to actually draw lines and circles. A subset of these Xlib functions will be described.

C.1.1 Setup

All Xlib drawing functions require a tool called a graphic context with which to draw, and a pixel map with which to draw onto. The following Xlib function will create a bitmap where each bit represents an actual pixel on the screen:

XcreatePixmap (display, drawable, width, height, depth);

The first parameter, *display*, represents the display name of the screen server. If, for example, the name of the workstation was grenada, then the display name might be "grenada:0". The second parameter, *drawable*, represents the application window which the pixel map must be modeled after. The width and height of the pixel map are specified by *width*, and *height*, in pixels. The final parameter, *depth*, defines the number of bits needed at each pixel to store the color of the pixel. The color of a pixel is usually stored as an 8-bit index into a color palette whose entries describe the Red, Green and Blue (RGB) shades required to generate the desired color. A pixel map will be returned by the function if successful.

The next function will create a graphical context or GC which defines the default color, thickness and style of the pen used to draw lines and circles. The style defines the continuity of the line (solid, dotted, dashed, etc...).

XcreateGC (display, drawable, valueMask, values);

The first parameter will be identical for all the Xlib functions described in this appendix. The second parameter will now represent the *drawable* onto which line and circles are to be drawn. This is the pixel map returned by the first function. It is possible to use the application window instead, however, if it is hidden by another, then the drawings will be permanently destroyed. The last two parameters allow the pen attributes to be set. *Value-mask* indicates which attributes will be changed and *values* is a structure of type *XGCValues* whose entries contain the new attribute values. An example is provided on how to change the foreground and background color attributes:

```
XGCValues values;
values.foreground = BlackPixel(XtDisplay(widget),1);
values.background = WhitePixel(XtDisplay(widget),0);
GC aGc = XCreateGC(XtDisplay(widget), thePixmap,
                  GCForeground | GCBackground, &values);
```

As can be seen in the example, the function returns a graphical context.

The white and black colors are always guaranteed to be the first and second entries in the current color palette. This is not true of the other colors. A desired color can be searched for in the color palette by name or RGB value and the index of the closest matching color will be returned. Alternatively, empty color palette entries can be allocated to hold new colors with the following function.

XAllocColorCells (display, colorPalette, contig, planeMasks, #OfPlaneMasks, newColorIndexes, #OfColorIndexes);

The second parameter represents the current color palette whose empty entries are to be allocated. The third, fourth, and fifth parameters are irrelevant for the purposes of allocating new entries. The sixth parameter, *newColorIndexes*, must represent an array of long integers where the indexes of the new color palette entries will be dumped. The final parameter represents the number of new palette entries desired.

Once new palette entries have been allocated, the RGB color values in the entries can be referenced by their indexes, but first the new entries must be filled with colors. The following function will fill in one entry.

XStoreNamedColor (display, colorPalette, colorName, entry, pixelFlags);

The third parameter, *colorName*, is a character pointer to the name of a color whose exact RGB values are to be placed in the *colorPalette* at index location *entry*. The last parameter, *pixelFlags*, must have the red, green, and blue flags set. An complete example is provided as follows:

```

unsigned long planeMask;
unsigned long colorIndex[18];
const char *color[] = {
    "hot pink", "navy blue", "cornflower blue",
    "dark slate blue", "medium blue", "light slate blue",
    "royal blue", "blue", "dodger blue",
    "deep sky blue", "light sky blue", "steel blue", "light blue",
    "powder blue", "cadet blue", "yellow", "slate blue"
};
Colormap colorPalette = DefaultColormap(XtDisplay(w),0);
XAllocColorCells(XtDisplay(widget),colorPalette,False,plane_mask,0, colorIndex,18);
for (unsigned i=0; i < 17; i++)
    XStoreNamedColor(XtDisplay(widget), colorPalette, color[i],
        colorIndex[i], DoRed | DoGreen | DoBlue);

```

The foreground color of the graphical context can be changed with the following function.

XSetForeground (display, gc, colorPaletteIndex);

The foreground color of the graphical context, *gc*, will be set to the *colorPalette* entry specified by *colorPaletteIndex*. In actuality, it is this index which is copied into every pixel drawn with this graphical context.

C.1.2 Drawing

Once a pixel map, graphing context, and colors have been setup, drawing can commence. However the pixel map will initially be filled with garbage and should be initialized. This can easily be done by drawing a fill rectangle which covers the entire pixel map area.

XFillRectangle (display, drawable, gc, x, y, width, height);

The foreground color in the *gc* will be copied to all the pixels within and including the border of the rectangle whose upper left hand conner is specified by *x* and *y* and whose dimensions are described by *width* and *height*.

A sequence of connected lines can be drawn using the following function.

XDrawLines (display, drawable, gc, arrayOfPoints, #OfPoints, mode);

This function can best be described using an example. The final parameter, *mode*, should be set to *CoordModeOrigin*.

```
XPoint *points = (XPoint *) calloc(3, sizeof (XPoint));
points[0].x = 10;
points[0].y = 10;
points[1].x = 20;
points[1].y = 10;
points[2].x = 15;
points[2].y = 20;
XDrawLines(XtDisplay(widget),aPixmap,aGc, points, 3, CoordModeOrigin);
```

The pixel map can be copied directly onto the screen using the following function:

XCopyArea (display, sourceDrawable, targetDrawable, gc, sourceX, sourceY, sourceWidth, sourceHeight, targetX, targetY);

The contents of the rectangle in *sourceDrawable* whose upper left hand corner is specified by *sourceX* and *sourceY* and whose dimensions are described by *sourceWidth* and *sourceHeight* will be copied to *targetDrawable* starting at *targetX* and *targetY*.

A description of the remaining Xlib functions can be found in Volume 2 of the X Window System, by O'Reilly and Associates.

C.2 X Toolkit Intrinsic

This layer is also called Xt and as such, all the functions implemented in this layer will start with Xt. Because all widget sets are based on Xt, they will have some common attributes or resources, such as the window which owns them, and the display which owns the window. In addition, all widgets have dimensions and many other resources which will not be discussed here. The following Xt functions described below are essential to invoking the XLib functions described in section C.1.

XtScreen (widget);

Given a widget from any widget set, this function will return platform dependent attributes which are necessary when drawing onto the physical screen.

XtDisplay (widget);

Given a widget from any widget set, this function will return the name of the display.

XtWindow (widget);

Given a widget from any widget set, this function will return the application window that the widget belongs to.

XtVaGetValues (widget, attributeFlag, varPointer, ..., attributeFlag, varPointer, NULL);

This is a general purpose function for accessing widget resources. In fact, it is so general that the second and third parameters may be repeated for different resources. It is therefore essential to mark the end of the parameter list with a NULL, otherwise a fatal error will occur when the application is executed. The parameter *attributeFlag* defines what resource is to be accessed and *varPointer* represents the address of the variable where the resource's value will be placed. An example is presented on how to extract the width and height of any widget.

Dimension screenWidth, screenHeight;

XtVaGetValues (w, XmNwidth, &screenWidth, XmNheight, &screenHeight,
NULL);

A similar function called *XtVaSetValues* allows widget resources to be changed. A more complete discussion can be found in volume 4 of the X Window System, by O'Reilly and Associates.

C.3 Widget Sets

Several different widget sets have been implemented across numerous unix platforms. Each widget set has its own style of appearance and functionality with respect to the

mouse. Some are easier to use than others. An attempt has been made to develop a standard widget set that users can become familiar with, which will be available on all platforms. The widget set called Motif is becoming an industry standard and is already supported by Sun and Hewlet Packard workstations. The graphical user interface of X applications may be manually written using the motif widget functions, or it may be automatically generated using an application development tool. The advantages of such tools is that widgets can be placed where desired on the application window using a mouse. Menus can be built, colors can be changed, and dialog boxes can be created and linked to menu items. When the graphical user interface appears the way the programmer wants it, the functions to create and manage the widgets will automatically be generated and placed in source files, usually in the form of c code. Empty callback functions will be generated for each widget to handle mouse events. When code is compiled and executed, the code which the programmer has added into the callback functions will be immediately executed when a widget is activated, such as selecting a menu item or button.

C.4 X Window Manager

The Common Development Environment (CDE) is one X window manager which is based on the Motif widget set. This window manager is used by Sun and Hewlet Packard workstations. The GUI of an application development tool which is shipped with the CDE can be seen in figure 55 on page 127. One widget of particular interest is the drawing area widget. The button under the heading “panes” in the figure with the colorful circle, rectangle and triangle, can be dragged into a blank application window to produce a drawing area for that application. A callback can be generated for the drawing area which will be invoked every time the window is partially or completely hidden. The widget will be passed in the callback parameter list and can then be used to invoke Xt and Xlib functions as presented in the examples in section C.1.1 and section C.1.2

Appendix D: Implementation details

This appendix provides a detailed explanation of the bdsII and XbdsII packages of bds97 as seen in figure 48 on page 84. The changes which have been made to the InCore parser, not already mentioned in chapter 6, consist only of 4 additional files which define a sub-class of ICvhdlParser called VHDLtoBDSII. This class as well as ICvhdlParser contain all the function calls to the cdfg and construct toolkits needed to create a hierarchy of CDFGs from the VHDL text file containing the behavioral circuit description.

D.1 The bdsII parts that generate CDFGs

This package is not actually a program but rather a set of toolkits which are designed to be used from other programs. There is no main function in the bdsII source code except for small programs to test each toolkit.

The source code is grouped into 38 directories, plus a recent add on directory called `x_draw` which will be described in section D.2. Many of the 38 directories contain the source code for utility classes such as dynamic arrays, lists, stacks, dynamic strings, etc... The directories which are responsible for creating and managing control/data-flow graphs are the `Element`, `CDFGData`, `cdfg_toolbox`, and `construct_toolbox` directories. Only these directories will be discussed.

D.1.1 The Element directory

This directory contains three building block classes, each of which are derived from a base class called `Element`. Each class is described as follows:

The Node class

Each instance of this class represents a node in a CDFG. The instance contains the name of the node and a list of pointers to terminal instances, one for each terminal owned by this node instance. A new pointer has been added which points to an instance of a Vertex class which contains attributes needed to draw the node. This pointer is initially null.

The Terminal class

Each instance of this class represents one terminal belonging to a node. The instance contains a pointer back to the node it belongs to and another pointer to the edge connected to the terminal. The instance also determines if the terminal is an input or an output. Only one edge may be connected to a terminal.

The Edge class

Each instance in this class represents one edge between two or more terminals. The instance contains a list of pointers back to the terminals connected to the edge. The edge instance determines the edge type (data or control, etc...) but not the direction. This is determined by the instance of each terminal. Normally only two terminals are connected to an edge, one input terminal and one output terminal. It is possible to have one output terminal connected to multiple input terminals. The opposite way around is not possible.

D.1.2 The graphContainer directory

The only class defined within this directory, GraphContainer, contains the actual elements for one control/data-flow graph. An instance contains a list of all nodes and a list of all edges used in a control-flow graph. In addition, this instance can keep track of a list of control steps for scheduling nodes, however the implementation of scheduling is incomplete.

There is an instance of a graphContainer for each control/data-flow graph in a hierarchy, however the nodes of the hierarchy itself consist of instances of a CDFGData class.

D.1.3 The CDFGData directory

One class is defined within this directory, the CDFGData class. An instance of this class contains pointers to all attributes related to one control/data-flow graph. A graphical representation of the CDFGData instance variables is shown in figure 54 on page 113. Some of

FIGURE 54. A graphical representation of the CDFGData class. The instance variable, theContainerNode, consists of nodes and edges which are connected to each other using pointers only and can therefore be drawn an infinite number of ways. The instance variable the Graph on the other hand determines the positions of each of the elements on a drawing area. The highlighted block contains instances of the graphical elements, which were added to CDFGData during the course of this thesis.

them are described as follows:

- theVariableTable: Consists of a tree of all variables which are used in this control/data-flow graph and any subCDFG's. Variables should be assigned to edges in a control/data-flow graph.
- theContainerNodes: This instance variable consists of a dynamic list of pointers to all

hiernodes nodes within the control/data-flow graph.

- **theSuperCDFG:** is a simple pointer pointing back to the control/data-flow graph of which this CDFGData is a hiernode of. If this CDFGData is the root of the CDFG hierarchy, then this instance variable is set to null.
- **theSubCDFGs:** This instance variable consists of a dynamic list of pointers to all control/data-flow graphs to which the hiernodes belong. The reciprocal is also true, each hiernode is itself aware of its CDFG owner.
- **theGraphContainer:** This points to an instance of the GraphContainer for this control/data-flow graph.
- **theGraph:** This pointer has been added during the course of this thesis for drawing purposes. It points to an instance of a Graph class which contains attributes needed to draw all nodes and edges in a control/data-flow graph. This pointer is initially null.

Most of the methods designed for the CDFGData class simply allow access or changes to be made to the instance variables described. The following method has been added for use by the front-end graphical interface to obtain an instance of class Graph.

createGraph();

This method creates an instance of a Graph class and returns a pointer to that instance. This method will be called by the front-end graphical interface to provide it with a class which recognizes and understands drawing messages. The Graph class can then carry out the messages from the data in the corresponding CDFGData instance.

D.1.4 The CDFG toolkit

This is the lower level of two bdsII toolkits which provides the In-Core parser with a mechanism to create CDFGs. The higher level toolkit will be discussed in section D.1.5. A toolkit consists of a group of functions which are compiled into a library so that they can be used in other programs. The function calls defined in the cdfg toolkit allow structural control/data-flows to be constructed on a step by step basis. Some of the 70 odd toolkit functions are described below. The remaining toolkit functions are not designed to be called externally, instead they are invoked by the following functions, or by those in the higher level toolkit.

cdfgt_init (void):

This function must be called before any other toolkit function.

cdfgt_create_CDFG (BString aName, GraphContainer* aGraph=NULL, CDFGData* aSuper=NULL);*

Given a name, this function will create and return a control/data-flow graph with a control input node and a control edge dangling from it. If a super CDFG is supplied, then a sub-CDFG will be returned.

*cdfgt_add_variable (const BString *aName, const VariableData *aVariable, CDFGData *aCDFG);*

Given the name of a variable, the variable itself, and a CDFG, this function will incorporate that variable into the CDFG.

*cdfgt_add_inport_node (CDFGData *aCDFG, const VariableData *aVarData);*

Given a CDFG, and the variable to be used, this function will create an inport node for the given variable along with an edge which will be returned. The node will automatically be bound to the current normal control-flow edge.

cdfgt_create_node (CDFGData aCDFG, Role_class_name rc, const VariableData* aVariableData);*

Given a CDFG, the node type (ex. Rc_const_node), and a variable, this function will create a constant node for the given variable and return the node itself. Based on the node type, a single output terminal will automatically be added to the node.

cdfgt_bind (aNode, aControlEdge);

Given a node and a control edge where both belong to the same CDFG, this function binds them together.

cdfgt_create_edge (CDFGData aCDFG, Role_class_name rc);*

Given a CDFG and the edge type (ex. Rc_read_write_dependency_edge), this function will return the created edge.

cdfgt_connect_node_edge (Node aNode, Edge* edge1, Edge* edge2,...);*

This function will then connect the output terminal of the created node with one end of the edge. This same function will connect all edges to a node with one input and output terminals, or two input terminals and one output terminal, etc...

The above functions can be used repeatedly to build up a control/data-flow graph consisting of many operation nodes binded to a simple straight flow control edge. Creating output nodes and completing a control/data-flow graph is handled by a more abstract toolkit.

D.1.5 The construct toolkit

This is the higher level of two toolkits. The toolkit functions described here are responsible for building behavioral constructs, wrapping up a control/data-flow graph, and then

connecting that isolated CDFG to a hiernode in another CDFG. This toolkit sits one layer above that of the *cdft* toolkit and as such will use some *cdft* functions to build the CDFG. Of all the functions described below, the first three are the only one unchanged from the original toolkit. The remaining functions have been altered or created during the course of this thesis to implement if, switch, and loop constructs. These functions were also modified to generate well ordered inport nodes, outport nodes, and hiernode terminals such that when drawn, a minimum number of crossovers between edges would occur, as described in section 4.4.3 on page 54.

ct_init (void)

This function must be called before any other toolkit function.

ct_create_top_level (void)

This function creates the root CDFG and returns it.

*ct_create_subprogram (CDFGData *aSuper)*

Given a CDFG, this function will create and return a subCDFG which will inherit all the variables from the superCDFG. After the subCDFG has been filled in, complete with inport and outports, it must be hooked up to a hiernode in the supercdfg.

ct_create_operation (CDFGData aCDFG, Role_class_name rc, Role_class_name rcOutEdge, Edge* left, Edge* right)*

This function can be used to create operation nodes more rapidly. Given the CDFG, the type of operation, the type of output edge, the left edge, and right edge, if applicable, this function will create the desired condition node, bind it to the control edge, connect the input edges, create an output edge, connect it to the operation node, and finally, return the output edge.

*ct_assign_variable (CDFGData *aCDFG, BString *sourceName, BString *destName)*

Given two variable names and the CDFGData where they exist, this function will assign the data-flow associated with the first variable to the second variable. Any data-flow associated with the second variable is lost.

*ct_resolve_constant (CDFGData *aCDFG, BString *varName, VariableData *newVarData)*

This function is used to resolve a variable, whose initial value cannot be resolved during parsing. If the name of the unresolved variable is *varName* then its initial value will be replaced with that in *newVarData*. This function is recursive and will be invoked with the same parameters for all sub CDFG's as well.

ct_wrap_up_cdfg (CDFGData* aCDFG);

Once a CDFG has been filled in using *cdft* toolkit functions, it must be wrapped up. This function will automatically create outputs for all dangling data and control edges which exist in the CDFG. Since only one type of output is allowed for each control, multiple control-flows of the same type will be merged together using a join node before being connected to an output control node. This function must be manually invoked to wrap up a behavioral construct.

ct_wrap_up_condition (CDFGData *condCDFG, Edge *condEdge)

This function will create a special data output for the given *condEdge* when wrapping up the CDFG so that the output can easily be recognized by the next three behavioral construct functions described below.

ct_create_if_then_else (CDFGData *aSuper, CDFGData *conditionBody, CDFGData *thenBody, CDFGData *elseBody);

As the name suggests, this function will create an if CDFG from start to finish. A generic if CDFG created from this function is shown in figure 21 on page 45 and figure 22 on page 46.

The parameters *conditionBody*, *thenBody*, and *elseBody* consist of subCDFGs of the first parameter, *aSuper*. All the subCDFGs must be filled in and wrapped up. The *conditionBody* CDFG must have exactly one data output and one normal control output. The other subCDFGs are optional. This function will create hiernodes for each of the subCDFGs and hook them up. A fork node will be connected from the *condition* hiernode. The fork outputs will be connected to the *else* and *then* hiernodes in that order. The if construct will be finished up with a join node if necessary. The filled in *superCDFG* will be returned.

ct_create_loop (CDFGData *aSuper, CDFGData *conditionBody, CDFGData *topBody, CDFGData *bottomBody, int hardware_flag);

This function is responsible for creating all types of CDFG loops. An generic example created from this function is shown in figure 24 on page 50.

The subCDFGs required in this function are referred to as *conditionBody*, *topBody*, and *bottomBody*. They must all be descendants of the first parameter, *aSuper*. The first one must adhere to the criteria specified in the *ct_create_if_then_else* toolkit function. The other subCDFGs are optional, but if included, must contain a normal control output. This function will initially create a normal join node, the right side of which the normal control-flow enters. The join node is followed by a hiernode representing the *topBody*, followed by a hiernode for the *conditionBody*, followed by a fork node. Only the left (*else*) side of the fork node will be connected a hiernode of the *bottomBody*. The right (*then*) side will become the new normal control-flow. The *bottomBody* hiernode will then loop back to the left side of the join node. The filled in *superCDFG* will be returned.

CDFGData ct_create_switch* (CDFGData *aSuper, CDFGData *conditionBody, DynamicArray<CDFGData> *caseBodies, CDFGData *defaultBody);

All switch CDFGs such as the generic one shown in figure 23 on page 48 are created by this function.

The second parameter, *conditionBody* consists of a subCDFG which must have a multi-valued data output node. Each value must correspond to one subCDFG in the list of CDFGs contained in the third parameter, *caseBodies*. The fourth parameter, *defaultBody*, is optional. If it is set to null then a blank cdfg will be created to handle invalid data values produced by *conditionBody*. All the subCDFGs must be descendants of the first parameter, *aSuper*. Hiernodes will be created for all the subCDFGs. The *conditionBody* hiernode will be connected to a fork join, the outputs of which will connect to every case hiernode, separated by join nodes used to merge in the normal control output from the previous case hiernode, if applicable. Any break control-flows will then be merged together in a join node. The output of the join node is then merged to the normal control-flow at the end of the switch construct. The completed switch cdfg will be returned.

Node ct_hookup (CDFGData *aSuper, CDFGData *construct, Role_class_name construct_role);*

A CDFG which has been filled in using one of the above three toolkit calls must be hooked up to a hiernode after it has been wrapped up. This function will create a hiernode in the CDFG represented by the first parameter, *aSuper*. The third parameter *construct_role*, will be used to determine the type of hiernode. The CDFG represented by the second parameter, *construct* will then be hookup to that hiernode.

D.2 The graphical elements

The c++ classes which visually display a cdfg as a pixel map are grouped under a sub directory of bdsII called *x_draw*. While these classes are not actually part of the bdsII internal representation, they are compiled as such because each instance of a CDFGData contains a direct instance variable to its equivalent graphical instance. This implies that the graphical elements must be pre-parsed before the CDFGData source code can be compiled. This file dependency could have been avoided through the use of hangers, rather than an instance variable, however this would have resulted in a slow drawing speed. The graphical elements consists of the classes Vertex, Arc and Graph. All of these classes were designed, developed and implemented during the course of this thesis to rapidly verify the correctness of the construct toolkit functions which generate if, switch, and loop con-

structs. By implication, the supporting toolkit functions as well as those in the cdfg toolkit are also verified. Each graphical class will be described in detail.

D.2.1 The Vertex

This class is responsible for displaying nodes. All nodes contain an instance variable pointing to their equivalent Vertex. Some of the instance variables of a Vertex are listed as follows.

- A set of two coordinates, the upper left hand corner, and lower right hand corner of the node.
- A flag to indicate if it has already been graphed in the bitmap or not.
- A pointer back to the actual node itself.
- A dynamic array of pointers to all the input terminals.
- A dynamic array of pointer to all the output terminals.
- The display label in the form of a character string.

In addition to a set of simple methods to access and change the instance variables mentioned, an addition set of support methods are needed to draw the vertex onto a bitmap. A few of these methods will be described.

getTermCoordinates (In_out anInOut, unsigned index, Point& aPoint);

Returns the coordinates of a selected input or output terminal as a Point. A Point is a simple c++ class with an x and a y instance variable. Terminal coordinates are automatically generated on the fly. The coordinates themselves are derived from both the given index in the second parameter and whether it is an input or output terminal as determined by the first parameter.

*getOtherTerminal (Terminal *aTerminal)*

Given a terminal, this method will follow the edge connected to it, retrieve and return the terminal at the other end. If there are multiple output terminals, then only the first one will be retrieved.

findLengthOfCriticalPath ();

This method consists of an algorithm which determines and returns an array of Vertices indicating a path from the node which is farthest away from the receiver instance (this). This search is possible because each instance of a Vertex knows all of its terminals, a terminal knows which node it is connected to, and finally, a node knows its equivalent Vertex.

The following next two methods deal with actually deciding where to place the node in the *pixel map*. The following algorithm is used to perform this. For any given Vertex whose node is to be graphed, all the Nodes connected to the input terminals of the given Vertex will be graphed first. This algorithm will execute recursively until a node without input edges has been reached. Loops are handled by treating input terminals connected to previously graphed Vertices as edgeless input terminals. Notice that Nodes will be drawn from the bottom up. This avoids the complication of handling exceptions where one output terminal is connected to two input terminals.

The process of graphing only refers to assigning coordinates to all the Vertices. The generation of a *pixel map* from the coordinates will be described as a drawing process. A list of coordinates, called the *frontier*, is used as a framework for assigning coordinates to Vertices. Each item in the array contains the coordinates of the next available position for that level. Vertices are separated into levels when they are drawn one under another. If there is a total of seven Vertices drawn vertically, then the highest vertex is drawn in level zero, and the last in level six. When coordinates are assigned to a Vertex, the frontier will automatically be updated with new unused coordinates.

Two more lists are used to keep track of Vertices and Arcs which have already been drawn. This information will be used for reducing the number of crossovers in routing.

A fourth list, called *dataFlowColors*, contains a list of data-flow variable names which have already been assigned unique colors. This allows all edges transporting values from a common variable to be drawn in the same color.

*graphNodeAt (Terminal *inTerminal, unsigned i, DynamicArray<Point> *frontier, unsigned level, DynamicArray<Vertex> *theVertices, DynamicArray<Arcs> *theArcs, DynamicArray<BString> *dataFlowColors);*

The purpose of this method is to graph the Node at the other end of the edge connected to *inTerminal*. This terminal belongs to the receiver Vertex. The second parameter *i* represents the index of the input terminal. The method actually responsible for graphing the Vertex will be described below. It is only called from this method. Once graphed however, this method will then generate an Arc to connect the respective terminals of both Vertices together. An Arc contains graphing attributes needed to draw its associated Edge. The

routing of the Arc will be dependant on obstacles which may exists between the two Vertices. The third, forth, fifth, and sixth parameters contain the lists described above.

*anchor (DynamicArray<Point> *frontier, unsigned level, DynamicArray<Vertex> *theVertices)*

Given the set of points making up the *frontier*, and the current *level* the receiver Vertex should be drawn at, this method will assign a pair of unused coordinates from the *frontier* to the Vertex. The Vertex itself will then be added to a list of anchored or drawn vertices called *theVertices* so that their position can be used in collision avoidance methods when drawing the Edges between the vertices.

*graph (DynamicArray<Point> *frontier, unsigned level, DynamicArray<Vertex> *theVertices, DynamicArray<Arcs> *theArcs, DynamicArray<BString> *dataFlowColors);*

This method actually graphs the Node associated with the receiver Vector by invoking the method *anchor*. The *graphNodeAt* method is then called for each of the input terminals in order of their index.

*reDraw (Display *aDisplay, Drawable aPixmap, GC aGc, float scale);*

Once a Vertex has coordinates, it can be easily drawn onto the pixel map referenced with *aPixmap*. The third parameter, *aGc*, supplies the graphical tool which is needed to actually draw onto the pixel map. The forth parameter *scale* represents the magnification factor that the Vertex should be drawn with. All vertices will have their input terminals evenly spaced out along the top edge, and their output terminals evenly spaced out along the bottom edge.

D.2.2 The Arc

This class is responsible for displaying the edges between nodes. Instances of this class are automatically generated while graphing Nodes. The edges between the Nodes to be graphed are used to generate corresponding arcs, however an Edge has no knowledge of its equivalent Arc. Some of the instance variables of an Arc are listed as follows.

- A list of points which, when connected together, represent a complete path between two terminals belonging to different Vertices.
- A pointer back to the corresponding Edge.
- The type of Edge, a control-flow or data-flow edge.
- The index of the color index in the color palette to be used when drawing this Arc onto a pixel map.

Besides the obvious get and set methods, a group of methods for determining and avoiding crossovers between drawn Arcs have been added. Some of these method will be described.

determinant (Point& a, Point& b, Point& c);

Given three points, this method will return the following values based on their orientation.

-1: Point *c* is to the right of the line segment *a, b*.

0: The three points form a linear line segment.

1: Point *c* is to the left of the line segment *a, b*.

intersect (Point& a0, Point &a1, Point& b0, Point& b1);

Given two line segments *a* and *b*, this method will determine if they intersect, and if so in what orientation. The intersection can be detected by calling the *determinant* method described above with all combinations of choosing three out the four parameters.

*intersect (Point& source, Point& destination, Vertex *aVertex);*

This more complicated version of the intersection will test for an intersection between a line segment and a rectangle which is defined by the two coordinates within a graphed Vertex.

*intersect (Point& source, Point &destination, DynamicArray<Point> *arcs);*

This method checks for an intersection between a line segment and a list of connected line segments of which an Arc is composed of.

*visible (DynamicArray<Vertex> *theVertices, DynamicArray<Arcs> *theArcs, Point& source, Point& destination);*

The line segment defined by a given *source* and *destination* point, will be tested with each Vertex in *theVertices*, and each Arc in *theArcs* for a possible intersection. Should an intersection occur with a Vertex, then an attempt will be made to form a detour around it. If an intersection occurs with an Arc instead, then that Arc will be followed up to its Vertex and the detour strategy will be applied. The type of detour strategy applied depends on whether the Vertex is an inport or outport node. If a successful detour is found, or if no intersection was detected, then True will be returned indicating that a visible or traversable path was found between the given points.

*pylons (Vertex *aVertex, Point &source, Point &destination, Point &upperLeft, Point &upperRight, Point &lowerRight, Point &lowerLeft);*

Given a vertex, this method will generate and return four pylons around the vertex which, when connected together, represent a non intersecting path around the vertex. The Points *source* and *destination* represent the line segment which currently intersects the vertex, or an Arc connected to the vertex.

The next two methods will determine the path around the vertex *aVertex*. The value of *i* in the fourth parameter indicates the orientation of Points *source* and *destination*. If *source* is to the left of the centre point of the given Vertex, and *destination* is to the right, then a clockwise detour around the Vertex will be attempted. If the points are oriented the opposite way around, then a counter-clockwise detour will be tried. If one fails, the other detour

will be tried anyway. If a successful detour is found, then the pylons used to perform the detour will be inserted between *source* and *destination*, and the list of points will be incorporated into the Arc. If no non-intersecting detour is found, then only *source* and *destination* will be incorporated into the Arc. If the vertex is neither an inport nor an outport node, then no detour is attempted.

detourUpAndOver (*DynamicArray*<*Vertex*> **theVertices*, *DynamicArray*<*Arcs*> **theArcs*, *Vertex* **aVertex*, *int* *i*, *Point*& *source*, *Point* &*destination*);

This method will be called by the *visible* method when the vertex is a inport node.

detourDownAndUnder (*DynamicArray*<*Vertex*> **theVertices*, *DynamicArray*<*Arcs*> **theArcs*, *Vertex* **aVertex*, *int* *i*, *Point*& *source*, *Point* &*destination*);

This method will be called by the *visible* method when the vertex is an outport node.

reDraw (*Display* **aDisplay*, *Drawable* *aPixmap*, *GC* *aGc*, *float* *scale*, *unsigned long* **colorIndex*);

Once a list of points have been generated describing the path of the Arc from its source to its destination, the Arc may be drawn onto the pixel map *aPixmap* using the drawing tool *aGc*. The fourth parameter, *colorIndex*, contains the color palette to be used when displaying the pixel map onto the screen. The drawing tool will be assigned the color index entry in the palette pointed to by an entry in *colorIndex*. The index of that entry is stored in an instance variable of the receiver Arc.

D.2.3 The Graph

All the attributes related to drawing a CDFGData onto a screen are grouped together into this class. Some of these attributes or instance variables are described as follows:

- The *screenWidth* and *screenHeight* which define the dimensions of the pixel map on which a CDFG will be drawn. The dimensions should at least be as large as the resolution of the computer monitor to permit full screen viewing.
- The scaling factor represents the magnification used to draw the Vertices and Arcs belonging to a CDFG, onto its pixel map.
- A pointer back to the corresponding CDFGData.
- A list of all Vertices belonging to the CDFGData.
- A list of all Arcs belonging to the CDFGData.
- A pointer to the previous CDFGData of which this CDFGData is a hiernode of. If the CDFGData is the root of the hierarchy, then this instance variable will be set to NULL.
- An color index table. This table consists of selected indexes to the current Xt color pal-

ette. The color values in the color palette of the selected indexes define the best shades of blue. Each unique CDFG variable will be assigned its own shade such that each variable can be identified on the screen by its shade. The first index in the color index table however points to an entry in the color palette which defines the color used to draw control edges. The second index points to an entry in the color palette which defines the color used to draw the labels. The remaining indexes point to the shades of blue.

The following instance variables are data types defined by the X window system. They are used by the methods in the Graph class to draw Vertices and Arcs onto a pixel map, and then to paint the pixel map into a drawing area owned by a window. A description of some of the functions defined by the X window system can be found in appendix B.

- A motif widget: This is a handle to the drawing area of a window on the physical screen.
- The pixel map: Because each pixel map allocates a large block of memory, excessive disk swapping occurs with only a small number of Graph instances. Each instance therefore points to the same pixel map. This implies that all Vertices and Arcs belonging to any particular CDFGData must be redrawn onto the pixel map every time that particular CDFGData is reselected. The time required to perform this function is negligible compared to the delay induced due to disk swapping.

Each CDFGData contains an instance variable which points to its corresponding Graph class. Some of the methods used by the Graph class are described below.

*findLengthOfCriticalPath (const DynamicArray<Node> *nodes, Vertex &farthestVertex);*
 Given a list of all the nodes belonging to the corresponding CDFGData for this Graph, this function will return a list of Vertices representing the longest path from an output node to an input node. The list itself is generated by invoking the Vertex method *findLengthOfCriticalPath*, for each input node and then returning only the longest list. The number of Vertices in the longest list also represents the maximum number of levels in the CDFG.

*graph (const DynamicArray<Node> *nodes);*

Given a list of all the nodes to be graphed, the method *findLengthOfCriticalPath* is first invoked to determine the maximum level of the graph. The output nodes and corresponding Vertices are then extracted and the *graph* method is invoked for each of the output Vertices. All Nodes and edges will then be automatically graphed from the bottom up.

trespassingNode (int x, int y)

This method determines if the coordinates *x* and *y* represents a location on the border of, or inside of a Vertex owned by the graph. If such a Vertex is found then its Node is returned, otherwise NULL is returned.

*selectGraph (Node *selectedNode)*

Assuming that *selectedNode* is a hiernode, then the CDFGData represented by the hiernode will be returned. If *selectedNode* is not a hiernode, then NULL is returned instead.

*reDraw (const char *backgroundColor);*

This method simply calls the *reDraw* method for all Vertices and Arcs owned by the Graph, thus redrawing the pixel map of the Graph. The parameter *backgroundColor* defines the color the pixel map should be initialized to before drawing the Vertices and Arcs.

The next four methods are indirectly invoked from the front-end user interface.

*initialize (Widget widget, unsigned long *colorIndex, Drawable aPixmap);*

This method is initially invoked on the root Graph. The parameters, all of which have been previously described, are stored in instance variables so that they do not need to be included in the next three methods. This method will then invoke the method *graph* described above and finally, recursively invoke this method for all sub Graphs of this one. Once the root Graph *initialize* method is complete, then all Graph instances will be ready to be drawn.

reSize (float scalingFactor);

This method updates the Graph instance with the latest *scalingFactor* chosen by the user. The *reDraw* method will automatically be invoked to regenerate the graph on the pixel map at the correct magnification. Lastly, the *rePaint* method will be invoked.

rePaint ();

Assuming that the graph has been drawn onto the pixel map, this method will copy the entire pixel map onto the drawing area of the window owned by the front-end GUI. The front-end GUI itself will determine how much, and which area of the pixel map will actually be visible to the user.

*input (XButtonEvent *event);*

This method is invoked by all mouse events which occur in the drawing area of the window owned by the GUI. If the left button on the mouse is released, the coordinates of the mouse position are sent to the *trespassingNode* method. The results are then fed into the *selectCDFG* method. If a new CDFGData is selected, then the Graph instance of the corresponding CDFGData is returned.

D.3 The front-end Graphical User Interface (XbdsII)

The front-end GUI is based on the X Window System, version 11, release 5[19]. This window system is a standard recognized by virtually all unix workstations. It consists of layer upon layer of X library subroutines starting with the X library at the most primitive level. The next layer consists of a higher level of subroutines called toolkits. Many different choices of toolkits exist, however the one used for this GUI is called the Xt Intrinsics library. The next higher layer consists of *widget* sets, of which there are multiple choices based on the Xt Intrinsics library. The one chosen here is called the Motif widget set. A brief description of the layers can be found in appendix B. Finally, at the top layer is the X window manager, which determines the general appearance, and functionality of windows. Window managers may also contain a X library of very high-level widgets not found in the basic widget sets. The window manager used to implement this GUI is called the Common Development Environment or CDE. It was chosen based on the fact that it appears to be the most widely used window manager across different unix workstations today.

The XbdsII GUI can be broken down into two parts, the application builder, and an externally defined source file to interface the c language calling convention used by the application builder with the c++ classes already described. Each part will be described separately.

D.3.1 The application builder

This is a Common Development Environment tool used to rapidly build graphical user interfaces. It consists of a window filled with configurable widgets which the user can select and drag into his application window. The main window of the application builder is

shown in figure 55 on page 127. The widgets also include menu bars, sub menus, popup

FIGURE 55. The CDE application builder.

menus, etc... When the visual appearance of the users' application window is ready, c language source code is automatically generated to implement the application. This process generates two principal source files and some utility files. The first principal source file is given the same name as the users application window, followed by c. This file contains some structure definitions followed by the function *main*. The main function initializes the widgets and then enters an endless main application loop, ready for mouse or keyboard events to occur. When an event does occur, the window manager decides where to send the event based on the location of the mouse. The mouse must be inside a widget area in order for that widget to receive the event. Events are sent to widgets via *callback functions*, which have been registered with the window manager before execution is passed onto the main application loop.

The second principal file contains *callback functions* for all the widgets in the users application window. The file is given the name of the main application window followed by “_stubs.c”. The callback functions have already been automatically registered with the window manager, however the action to be taken by the widget is up to the programmer. and therefore the functions are empty. Once the callback functions are completed, the entire directory can be compiled with links to external toolkits. A single self contained executable will be generated, ready to be run.

A description is presented below of the callback functions. All callback functions have the same parameter list. The first parameter represents the widget who's callback function is being invoked because of an event within the focus of the widget. The second parameter contains a data structure which is setup by the programmer when registering the callback function with the window manager. Usually this parameter in the registration function is left null if the programmer has no data to pass into a callback function. This is the case for all the callback functions described below. The third parameter consists of a pointer to a system structure which contains information about various attributes such as the current position of the mouse at the time the event occurred.

Open (Widget widget, XtPointer clientData, XtPointer callData);

This is the first callback function which should be invoked. After the user has opened the file dialog box by selecting the menu item “open” in the file menu, and a VHDL source file has been selected, then this callback will be executed. The selected filename is retrieved from a structure member in the *callData* parameter. The filename will then be passed to an externally defined function called *createRootCDFG* which will parse the source file and generate a CDFG hierarchy. A second externally defined function called *initializeCDFGGraphics* is then invoked to generate instances of c++ graph classes for each CDFG in the hierarchy. The drawing pane widget is passed to the second function as a parameter to provide the instances of class Graph with a means to access the drawing pane.

RePaintPixelmap (Widget widget, XtPointer clientData, XtPointer callData);

This callback function is invoked whenever the XbdsII window is hidden by another, minimized, maximized, moved off screen, etc... and is then brought back into view. The first parameter represents the drawing pane onto which a pixel map may be painted. The drawing pane, like everything else in the window, is a widget. The only action taken by the callback is to invoke an externally defined function called *rePaintCDFGGraphics*.

ReSizePixmap (Widget widget, XtPointer clientData, XtPointer callData);

This callback function is invoked whenever the value inside a spinBox widget is altered. The first parameter contains the spinBox widget. This widget is defined by the CDE and consequently, information on its properties can be found in the online help applications. The third parameter, *callData*, is used to determine the actual value in the spinbox widget. The value is then sent as a parameter to an externally defined function called *ReSizePixmap*.

Input (Widget widget, XtPointer clientData, XtPointer callData);

This callback function is invoked for any mouse events which occur within the focus of the drawing pane. The first parameter contains the drawing pane widget. The third parameter, *callData*, is used to determine the position and type of the event. This information is then passed as a parameter to an externally defined function called *CDFGGraphicsInput*.

Exit (Widget widget, XtPointer clientData, XtPointer callData);

When exit is selected from the file menu bar of the application window, this callback function will be invoked. The first parameter contains the main window widget. This callback will execute one externally defined function called *removeCDFGGraphics* to destroy the instances of the c++ graphing class, and another externally defined function called *destroyRootCDFG* to destroy all the CDFG's themselves and the toolkits. Finally, a simple c exit function call will terminate the entire application program.

D.3.2 The external interface file

This file resides in its own directory under bdsII called Instantiation. Besides defining all the external functions needed to allow the graphical user interface to communicate with the c++ classes, this file also instantiates all templates used by the bdsII toolkits. By instantiating the templates as late as possible in the compilation process, changes may be made to the toolkit functions themselves without necessarily having to recompile this file. This is one reason why this file is located under the bdsII directory. Another reason is that it must be compiled using a c++ compiler to permit the external functions to invoke methods within the Graph class itself. This cannot be performed directly from the callback functions described above since they should be compiled using a non object oriented c compiler. The file itself does not contain any class definitions, only self contained functions which will be compiled as ordinary c functions due to an extern "C" compiler direc-

tive. This will allow the functions to be invoked from the callback functions. Each function is briefly described below.

createRootCDFG ();

This c function simply calls an externally defined c++ function called *generate-RootCDFG()*. This function will be defined in a source file in the In-Core directory where a VHDL input file is parsed into a CDFG hierarchy. The returned root CDFG will be stored in a global variable of the external interface file.

initializeCDFGGraphics (Widget w);

Given the drawing area widget, this function will extract the display handle and its default color palette. A new pink color and a variety of blue shades will be added to the color palette. The color palette indexes of which will be stored in a color table. The dimensions of the drawing area will also be extracted from the widget and used to create a single pixel map. An instance of class Graph will be created for the rootCDFG. The drawing area widget, color table and pixel map will be sent as parameters to the instance via an *initialize* method. Finally, a default scalingFactor, which is also a global variable, will be sent with the Graph class *reSize* method to force the rootCDFG to be displayed onto the drawing pane. The instance of class Graph will be stored in an instance variable which keeps track of the current instance of class Graph being displayed onto the drawing area.

reSizeCDFGGraphics (float scalingFactor);

The new scalingFactor is sent to the current instance of Graph by invoking its *reSizeCDFGGraphics* method. This will cause the CDFG contained in the current instance of Graph to be redrawn onto the drawing area in the proper magnification.

rePaintCDFGGraphics ();

This function simply invokes the *rePaint* method for the current instance of class Graph.

*CDFGGraphicsInput (XButtonEvent *event);*

Given a mouse event structure, an attempt will be made to retrieve the Graph represented by the node in which the mouse event was triggered by invoking the *input* method on the current instance of class Graph. If a new instance of class Graph is returned, then that will become the new current instance of class Graph. The *reSize* and *rePaint* method will be invoked to force the selected Graph to be displayed onto the drawing pane.

removeCDFGGraphics ();

This function calls the *removeGraph* method for the rootCDFG.

destroyRootCDFG ();

This function invokes an externally defined c++ function called *removeRootCDFG* and passes the rootCDFG as a parameter. The function, which is defined in a source file in the In-Core directory, will free up all memory used for all instances associated with the CDFG hierarchy.

Appendix E: Building a CDFG using toolkit calls

The following appendix will demonstrate how the toolkit functions described in appendix D can be used to construct a control/data-flow graph. The bdsII toolkit functions described in sections E.1 through E.12 already existed before this thesis was started.

E.1 Initialization

Three toolkits must be initialized before they are used, the cdfg toolkit, the construct toolkit, and the hanger toolkit. This last toolkit has not been described in chapter 4 because it is only needed internally by the cdfg toolkit. The code shown below performs the initialization of all three toolkits.

```
ht_init();  
cdfgt_init();  
ct_init();
```

E.2 The root CDFG

A CDFG hierarchy is created from the top down. This means that the root of the hierarchy must be created first, then its children etc... A blank root CDFG with one dangling normal control-flow edge from a default control inport node can be created using the following function.

```
CDFGData *rootCDFG = ct_create_top_level();
```

E.3 Declaring variables

Before any edges or nodes can be created and added, variables to contain the values being passed along the edges between nodes must be declared as follows:

```
VariableData *var1 = new VariableData();
```

A name can be assigned to a Variable by passing the BString of the name to a set method. BString is expected by toolkit parameters instead of char *.

```
var1->setVarName (new BString ("My first variable"));
```

The address style of the variable must then be set. There are four possibilities. An example of the set method is then shown:

A_register: The variable is a register.

A_local: The variable is local. (This may imply that it will be stored in a stack.)

A_param: The variable is a parameter. (This implies that it will be stored in a stack.)

A_global: The variable is global, meaning that it will be stored in memory.

```
var1->setAddressStyle (A_local);
```

The type of data to be stored in the variable is then set. The five possibilities are listed below, followed by an example:

- **IntType** (BString, bitWidth, boundary alignment): This type can represent unsigned, one's complement, two's complement and magnitude integers. The first parameter must contain the name of the variable. The second parameter expects the number of bits required to represent the maximum value. And the third parameter represents the bit boundary in memory where each value must be aligned with. For example for a 32-bit value, the boundary alignment should be every fourth byte.
- **FloatType** (BString, bitWidth, alignment, radix, mantissa, exponent). This type can represent most floating point numbers. The third parameter defines the base of the exponent. This is usually 2, 8 or 10 for decimal. The forth parameter represents the size of the mantissa in bits. The fifth parameter must contain the size of the exponent in bits.
- **FixedType** (BString, bitWidth, alignment, decimalPlace). This type is used to represent numbers with a fixed decimal point which can be implemented in hardware. The forth parameter specifies which bit will contain the decimal point.
- **CharType** (BString, bitWidth, alignment). This type is self explanatory. The bitWidth and alignment should be 8.
- **PointerType** (BString, bitWidth, alignment, ptrType). This type is a pointer which points to another variable, whose type is given in the forth parameter. Any of the above described types may be used including another pointerType.

```
var1->setType (new IntType(var1->getVarName(),4,4));
```

The initial value of a variable will always be stored as a BString. It can be set as follows:

```
var1->setInitialValue (new BString("0"));
```

The variable can be incorporated into a CDFGData instance as shown. The variable will automatically be inherited by sub CDFGData's.

```
cdfgt_add_variable (var1->getVarName(), var1, rootCDFG);
```

E.4 Creating a sub CDFG

A sub CDFG can be generated with the following construct toolkit call:

```
CDFGData *subCDFG = ct_create_subprogram(rootCDFG);
```

The resulting CDFGData will also contain a dangling normal control-flow edge from a default control inport node. That dangling edge can be accessed through the parser Variable Table (parserVT) of the CDFGData.

E.5 The Parser Variable Table.

There are five entries in a parserVT, one for each kind of control-flow. Each entry keeps track of the current control-flow and any associated data-flows. The table index constants needed to access the desired kind of flow are listed as follows:

Data_flow_Table: The normal control and data-flow.
Break_flow_table: The break control and data-flow.
Continue_flow_table: The continue control and data-flow.
Return_flow_table: The return control and data-flow.
Goto_flow_table: The goto control and data-flow.

Once the desired table entry has been chosen, the method *getCntlEdge* will return the dangling control-flow edge. An example is presented.

```
Edge *theCntl= subCDFG->getParserVT(Data_flow_table)->getCntlEdge();
```

E.6 Creating Nodes.

Additional inport nodes may be created using the following CDFG toolkit call:

```
Edge *edge1 = cdfgt_add_inport_node(subCDFG, var1,1);
```

The second parameter determines which variable will hold the value that has been passed from the superCDFG, in this case, rootCDFG. An edge will automatically be created and connected to the output terminal of the inport node. That edge will then be returned. The variable in the second parameter will be associated with the returned edge. The third

parameter is treated as information only and is usually used to store the line number of a hardware language which invoked the above CDFG toolkit call. All inport nodes will automatically be bound to the normal control-flow, since this is the only kind of inport control-flow allowed into a CDFG.

Conditional nodes are nodes which expect two input edges, and output a boolean edge. The value of the boolean variable associated with the edge is dependant on the condition operation selected to compare the inputs. A list of possible conditional operations is shown below. The node will automatically be bound to the normal control-flow.

Rc_eq_node: Compares the values of the input edges for equality.
Rc_ge_node: Outputs True if the left input is greater or equal to the right input.
Rc_gt_node: Outputs True if the left input is greater than the right input.
Rc_le_node: Outputs True if the left input is less or equal to the right input.
Rc_lt_node: Outputs True if the left input is less than the right input.
Rc_ne_node: Outputs True only if the value of both input edges are different
Rc_and_node: Outputs the boolean result of ANDing the left and right inputs.
Rc_nand_node: Outputs the boolean result of NANDing the left and right inputs.
Rc_nor_node: Outputs the boolean result of NORing the left and right inputs.
Rc_or_node: Outputs the boolean result of ORing the left and right inputs.
Rc_xor_node: Functionally identical to Rc_ne_node.
Rc_xnor_node: Functionally identical to Rc_eq_node.

The remaining operations are listed below.

Rc_abs_node: Outputs the absolute sign of the single input.
Rc_add_node: Adds the left and right inputs and outputs the sum.
Rc_sub_node: Subtracts the right input from the left input and outputs the difference.
Rc_left_rotate_node: Left rotates the left input by X bits as specified by the right input.
Rc_left_shift_node: Left shifts the left input by X bits as specified by the right input.
Rc_right_rotate_node: Right rotates the left input by X bits as specified by the right input.
Rc_right_shift_node: Right shifts the left input by X bits as specified by the right input.
Rc_div_node: Divides the left input by the right input and outputs the quotient.
Rc_mod_node: Divides the left input by the right input and outputs the modulus.
Rc_rem_node: Divides the left input by the right input and outputs the remainder.
Rc_mul_node: Multiplies the left and right inputs and outputs the product.
Rc_sqr_node: Outputs the square of the single input.
Rc_u_minus_node: Unary minus.
Rc_addressof_node: Outputs the address where the input value is stored.
Rc_bitwise_and_node: Outputs the bitwise ANDing of both the left and right inputs
Rc_bitwise_not_node: Outputs the bitwise inverse of the single input.
Rc_bitwise_or_node: Outputs the bitwise ORing of both the left and right inputs
Rc_bitwise_xor_node: Outputs the bitwise XORing of both the left and right inputs
Rc_not_node: Outputs the boolean result of ANDing the left and right inputs.

There are two types of output edges used when creating nodes. The difference between them is described in section E.7. The type shown in the example, `Rc_read_write_dependency_edge`, must have a variable associated with it. This will be shown in section E.9

```
VariableData *var2 = new VariableData();
var2->setVarName (new BString ("My second variable"));
var2->setAddressStyle(A_local);
var2->setType (new IntType(var2->getVarName(),4,4));
var2->setInitialValue (new BString("0"));
cdfgt_add_variable(var2->getVarName(), var2, rootCDFG);
Edge *edge2 = cdfgt_add_inport_node(subCDFG, var2,1);
Edge *bool = ct_create_operation(subCDFG,Rc_eq_node,
                                Rc_read_write_dependency_edge,edge1, edge2);
```

The construct toolkit function `ct_create_operation`, as shown in the example, can be broken down into more primitive CDFG toolkit functions. The following CDFG toolkit function will only create and return the node itself.

```
Node *aNode = cdfgt_create_node(subCDFG,Rc_eq_node);
```

The node must then be bound to a specific control-flow using the following `cdfg` toolkit function. The second parameter may be any kind of control-flow edge.

```
cdfgt_bind (aNode, theCntl);
```

Section E.7 and E.8 will show how this node can be connected to edges

E.7 Creating edges

The next `cdfg` toolkit function will create isolated edges which can then be connected to isolated nodes. A list of possible edge types is presented. The first two types are data edge types only, followed by an example.

Rc_data_edge: Transfers a value between terminals, without storage.

Rc_read_write_dependency_edge: Transfers a value between terminals by writing and the reading from a shared storage device.

Rc_control_edge: Used for any kind of control-flow.

The second edge type, `Rc_read_write_dependency_edge`, requires a variable to be associated with it to allow the value represented by the data edge to be stored. This will be discussed in section E.9 The following CDFG toolkit function is used to create an edge.

```
Edge *sumEdge = cdfgt_create_edge(subCDFG,  
                                Rc_read_write_dependency_edge);
```

E.8 Connecting Nodes and Edges

One generic toolkit function exists for connecting edges to all terminals of a node simultaneously. The first parameter consists of the Node itself, followed by each of the incoming edges, one parameter at a time, and then one or more output edges. The number of edge parameters is dependant on the number of terminals on the node, however the edges to be connected to input terminals are always listed first. The following example demonstrates how two incoming edges, and one outgoing edge are connected to the `Rc_eq_node` created above:

```
cdfgt_connect_node_edge(aNode, edge1, edge2, sumEdge);
```

The toolkit functions described so far are sufficient to produce control/data-flow graphs which are simple data-flow graphs. Outport nodes will be automatically created using the control and corresponding data-flow in each entry of the Parser VT.

E.9 Making the association between an edge and its variable.

This section only applies to edges of type `Rc_read_write_dependency_edge`. The CDFG toolkit function `cdfgt_set_variable_data`, shown in the example below, can be used to explicitly associate a variable with an edge which has been created, or returned from the construct toolkit function `ct_create_operation`. The CDFG toolkit function `cdfgt_add_inport_node` automatically makes the association since a variable is passed to it as a parameter.

```
VariableData *var3 = new VariableData();  
var3->setVarName (new BString ("The computed answer"));  
var3->setAddressStyle(A_local);  
var3->setType (new IntType(var3->getVarName(),4,4));
```

```
var3->setInitialValue (new BString("0"));  
cdfgt_add_variable(var3->getVarName(), var3, rootCDFG);  
cdfgt_set_variable_data (sumEdge, var3);
```

The variable associated with an edge can also be retrieved as follows:

```
VariableData *var3 = cdfgt_get_variable_data(sumEdge);
```

The association is implicitly performed in section E.10

E.10 Updating entries in the ParserVT

The current control-flow edge in the normal flow entry of the Parser VT is automatically updated, however the others are not. To ensure that dangling data-flow edges connected to output terminals of nodes are eventually connected to output nodes, the dangling edges and the variable represented by them can be registered with the ParserVT.¹ This can be done using the ParserVT method addDependency as shown below:

```
subCDFG->getParserVT(Data_flow_table)->addDependency(var1, sumEdge);
```

By registering the value contained in sumEdge with the variable var1, its original value as specified in its inport node will be overwritten by the new value generated by the Rc_eq_node. The old value however will no longer be needed.

E.11 Wrapping up a cdfg

The process of wrapping up a CDFG involves merging two or more instances of any kind of control and corresponding data-flow together, using a join node if necessary, such that there is only one flow of each kind. Each of these flows will then be automatically terminated with a control output node and corresponding data output nodes. The following construct toolkit function will perform this with subCDFG.

```
ct_wrap_up_cdfg(subCDFG);
```

1. This applies only to edges of type Rc_read_write_dependency_edge. Edges of type Rc_data_flow have no variable and as such their dangling end must be connected to another operation node.

E.12 Generating a postscript file

In addition to the graphical user interface, a CDFG may alternately be viewed by creating a *postscript* file of a specific CDFGData, and then viewing the file using a postscript viewer. The postscript file generating utility is very simple however, and will only work well for very simple CDFG's of no more than 10 nodes including inports and outports. There is no attempt to reduce the number of crossed edges, in fact, because terminals are not drawn, it is very likely that multiple edges will be drawn one on top of another. Furthermore, this utility will hang if an attempt is made to generate a postscript file of a cdfg containing loops. The cdfg toolkit function expects two parameters, an instance of the CDFGData, and the filename used to store the postscript file under. The extension ".ps" will automatically be added to the filename. An example of a postscript file is shown in figure 25 on page 53.

```
cdftg_draw (subCDFG, "subcdfg");
```

E.13 Creating behavioral constructs

The toolkits described here were implemented during the course of this thesis.

Once a CDFG has been wrapped up, it can be used along with other wrapped CDFG's to generate a behavioral construct. An example of how to invoke each type of behavioral construct is presented below. The first parameter in each function contains a blank CDFG which has previously been created using the function *ct_create_subprogram*. Hiernodes will be created for each of the CDFG's in the subsequent parameters and inserted into the blank CDFG. The other CDFG's must therefore be subCDFG's of the first. Once the behavioral construct is complete, its CDFG should be wrapped up.

E.13.1 The if construct

This construct expects three wrapped CDFG's, a conditionCDFG which has only a boolean data output and a normal control output, a thenCDFG, and an elseCDFG. Either one of the last two are optional and may be replaced with NULL, but not both.

```
ct_create_if_then_else (ifCDFG,condCDFG,thenCDFG,elseCDFG):
```

An example of an if construct is shown in figure 22 on page 46.

E.13.2 The switch construct

The switch construct expects a wrapped conditionCDFG and a list of case CDFG's. The default CDFG is optional and may be replaced with NULL. The only data output of the conditionCDFG must supply a value for every CDFG in the case list, starting with the value 0. If the conditionCDFG returns a value beyond the case list, then the default CDFG will be called. Any break flows from the list of caseCDFG's will automatically be merged to the normal flow after the defaultCDFG. An example is provided showing how a caseList can be generated and used:

```
DynamicArray<CDFGData> caseList;  
caseList.add(case1CDFG);  
caseList.add(case2CDFG);  
caseList.add(case3CDFG);  
ct_create_switch (switchCDFG, condCDFG, &caseList, defaultCDFG);
```

An example of a switch construct is shown in figure 23 on page 48.

E.13.3 The loop construct

This construct uses three wrapped CDFG's, a conditionCDFG identical to the one used in the if construct, a topCDFG which will be executed before the condition is tested, and a bottomCDFG which will be executed after the condition is executed. Any continue flows from the last two cdfgs will be merged with the normal flow just before the loop is repeated and any break flows will be merged with the normal flow after the loop has terminated. The topCDFG or bottomCDFG is optional, and may be replaced with NULL. but not both. An example of the construct toolkit call is shown:

```
ct_create_loop (loopCDFG, condCDFG, topCDFG, bottomCDFG);
```

An example of a loop construct is shown in figure 24 on page 50.

E.14 Generating a hiernode of a CDFG

All of the behavioral constructs described above automatically generate hiernodes for each of the wrapped CDFGs supplied in the parameters and then hook up the hiernodes into the super CDFG of the wrapped CDFGs by binding them to the current normal control-flow. The data input terminals on each hiernode are automatically connected to data-flow edges which have been registered with the ParserVT of the super CDFG. If a data input terminal cannot be matched up with an existing data-flow, then a new inport node in the super-CDFG is automatically created to provide a data-flow to that input terminal on the hiernode. The data and control-flow from all output terminals of the hiernode are automatically registered with the ParserVT so that the next hiernode may then be hooked up directly, or else the superCDFG wrapped up.

The construct toolkit function in the example below allows a hiernode to be created for a wrapped up behavioral CDFG itself, and hooked up to the superCDFG of the behavioral CDFG. The first parameter must contain the CDFG where the hiernode is to be added, the second parameter represents the wrapped up CDFG, and the third parameter specifies the desired type of hiernode to be generated. Each type of hiernode in the following list identifies the behavior of the CDFG it represents.

- Rc_general_df:** Represent any general CDFG
- Rc_function_call:** Represents a function.
- Rc_inline_asm:** Represents a block of in line assembly code.
- Rc_loop:** Represents a behavioral loop construct.
- Rc_top_loop_body:** Represents the top CDFG in a loop construct.
- Rc_bottom_loop_body:** Represents the bottom CDFG in a loop construct.
- Rc_hw_loop:** Represents a hardware loop.
- Rc_if:** Represents a behavioral if construct.
- Rc_condition:** Represents a condition CDFG.
- Rc_then:** Represents the left side of an if construct.
- Rc_else:** Represents the right side of an if construct.
- Rc_switch:** Represents a behavioral switch construct.
- Rc_case:** Represents one case in a switch construct.
- Rc_default:** Represents the extra case in a switch construct where the rejected selections go.

```
Node *hierNode = ct_hookup (rootCDFG, switchCDFG, Rc_switch);
```

E.15 Termination

The toolkit functions described above will enable a designer to generate a hierarchy of control/data-flow graphs to describe both control dominated and data intensive circuits. Once the designer is finished with them, they should be properly removed from memory using the following destructive toolkit functions:

```
ct_term();  
cdfgt_empty (rootCDFG);  
cdfgt_term();  
ht_destroy();
```

Notice that these functions are called in the opposite order from their corresponding initialization functions. The second destructive function frees up the memory allocated by the rootCDFG itself. This includes the Nodes, Terminals, and Edges owned by the CDFG. Since this function is recursive, the same will automatically be done to all subCDFGs, etc...

Master index

A

- ALAP 31
- algorithmically 9
- Arc 121
- architecture 87
- ASAP 31
- ASIC 30

C

- callback 110, 127
 - Exit 129
 - Input 129
 - Open 128
 - RePaintBitmap 128
 - ReSizeBitmap 129
- callback function 128
- CDFG 114
- cdfg toolkit
 - cdfgt_add_inport_node 115, 133
 - cdfgt_add_variable 133, 135, 137
 - cdfgt_bind 115, 135
 - cdfgt_connect_node_edge 115, 136
 - cdfgt_create_CDFG 115
 - cdfgt_create_edge 115, 136
 - cdfgt_create_node 115, 135
 - cdfgt_draw 138
 - cdfgt_empty 141
 - cdfgt_get_variable_data 137
 - cdfgt_init 114, 131
 - cdfgt_set_variable_data 137
 - cdfgt_term 141
- cdfg types
 - CharType 132
 - FixedType 132
 - FloatType 132
 - IntType 132
 - PointerType 132
- cdfg_toolbox 111
- CDFGData 88
- cdfgt_add_variable 115
- cdfgt_init 114
- concurrency 75
- concurrent 11, 76
- concurrently 9
- construct toolkit
 - ct_assign_variable 116
 - ct_create_if_then_else 117, 139
 - ct_create_loop 117, 139
 - ct_create_operation 116
 - ct_create_subprogram 116, 133
 - ct_create_switch 117, 139
 - ct_create_top_level 116, 131
 - ct_hookup 118, 140
 - ct_init 116, 131
 - ct_resolve_constant 116
 - ct_term 141
 - ct_wrap_up_cdfg 117, 137
 - ct_wrap_up_condition 117
- construct_toolbox 111

- control flow 11, 37, 41
- control step 12, 14, 15, 32
- control steps 12, 31
- createGraph 114
- critical path 18, 31, 33

D

- data dependencies 10, 12
- data flow 39
- data-flow 10
- determinant 122
- distribution graph 32

E

- edges 38
- encapsulated 95
- entity 87
- external
 - CDFGGraphicsInput 129, 130
 - createRootCDFG 128, 130
 - destroyRootCDFG 129, 130
 - initializeCDFGGraphics 128, 130
 - removeCDFGGraphics 129, 130
 - rePaintCDFGGraphics 128, 130
 - reSizeCDFGGraphics 130

F

- FFT 8
- FIR 19
- force 33
- frontier 120, 121
- FSM 12
- Functional unit 14
- functional unit 2, 36
- functional units 15, 16, 33

G

- Graph 123
 - findLengthOfCriticalPath 124
 - graph 124
 - initialize 125
 - input 125
 - rePaint 125
 - reSize 125
 - selectGraph 125
 - trespassingNode 124
- graphContainer 112

H

- hangers 118
- hiernode 37, 51, 75, 97, 118, 129, 140
- hiernodes 76, 114
- histogram 32

I

- ICbase 85
 - ICbitnet 89
 - ICconnection 89

- ICdesign 86
- ICexprData 89
- ICinstance 88
- ICinterface 89
- IClibrary 86
- ICmodule 87
- ICnet 88
- ICparameter 88
- ICport 88
- ICterminal 89
- ICvhdIBinary 90
- ICvhdIChar 90
- ICvhdIInteger 90
- ICvhdIPhysConstant 90
- ICvhdIReal 90
- ICvhdIString 90
- In-Core
 - architecture 30
 - entity 30
- internal representation 4

J

- join node 42

L

- lexical analyzer 85
- list scheduling 14
- logic minimization 20, 21

M

- Mealy 18
- Minimization 14
- mobile 32, 33
- Moore 18
- multi-cycling 12
- Multi-level logic 21
- Mutually exclusive 18

N

- net list 1, 16, 17, 20
- nodes 37

P

- pixel map 120, 124
- postscript 138
- precedence edge 2

R

- Register-transfer level 21
- resource 12, 14, 16, 31
- Resource constrained 14
- Retiming 18
- ROM 20

S

- schedule 14
- scheduling 14, 15
- sequential 75
- Storage elements 14
- storage elements 19
- syntactic groupings 85

T

- terminals 37
- Time constrained 14
- time constrained 31
- Two level logic 21

V

- verilog 88
- Vertex 119
 - anchor 121
 - determinant 122
 - detourDownAndUnder 123
 - detourUpAndOver 123
 - findLengthOfCriticalPath 119
 - getOtherTerminal 119
 - getTermCoordinates 119
 - graph 121
 - graphNodeAt 120
 - intersect 122
 - pylons 122
 - visible 122
- VHDL 23, 30
 - architecture 24
 - Component 24
 - Component instantiation statements 25
 - Components 24
 - Concurrent statements 25
 - Configuration specifications 25
 - entity 24
 - Nets 24
 - Ports 24
 - Process statements 25
 - Signal 24

W

- widget 126

X

- x_draw 111

Y

- yypostParse 86

References

1. S. Amellal, B. Kaminska, "Scheduling of a Control/data-flow graph", (*Proceedings*) *1993 IEEE International Symposium on Circuits and Systems*. IEEE, May 1993, pp.1666-9 vol. 3.
2. Lin Chi-Ho, Chong Jong-Wha, "A scheduling algorithm for conditional resources sharing consideration", *Journal of the Korean Institute of Telematics and Electronics*. Korea Institute of Telematics & Electronics, Feb 1996, pp.196-204.
3. T.C. Lee, N. K. Jha, W. H. Wolf, "A conditional resource sharing method for behavioral synthesis of highly testable data paths", *Proceedings. International Test Conference 1993*. IEEE, 1993, pp.744-53.
4. F. F. Hsu, E. M. Rudwick, J. H. Putel, "Testability insertion in behavioral descriptions", *Proceedings. 9th International Symposium on System Synthesis*. IEEE Comput. Soc. Press., 1996, pp.139-44.
5. Wang Qingsheng, Xue Hongxi, Su Ming, Bian Jinian, "Behavioral description in VisualVHDL and its implementation", *1995 4th International Conference on Solid-State and Integrated Circuit Technology. Proceedings*. IEEE, 1995, pp.361-3.
6. Ulrich Lauther, "Introduction to Synthesis", *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992, p. 4.
7. Cary Ussery, Carl Schaefer, Roger Lipsett, "VHDL". Intermetrics. 1989, pp.1-2.
8. "In-Core Reference Manual". Functionality Inc., September 1995. (BDS Volume 6A)
9. Sabine März, "High-Level Synthesis", *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992, pp. 162-167
10. Wolfgang Rosenstiel, Heinrich Kramer, "Scheduling and Assignment in High Level synthesis", *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1992, pp.359-361.
11. Kazutoshi Wakabayashi, "Cyber: High Level Synthesis System from Software into ASIC", *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1992, pp.129-141.
12. David C. Ku, Giovanni De Micheli, "High Level Synthesis of ASICs Under Timing and Synchronization Constraints". Kluwer Academic Publishers, 1992, pp.61-68.
13. D. E. Thomas, T.E. Fuhrman, "Industrial Uses of the System Architect's Workbench", *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1992, p. 312.
14. Cary Ussery, Carl Schaefer, Roger Lipsett, "VHDL". Intermetrics, 1989, pp.136-147.
15. Rumi Zahir, Wolfgang Fichtner, "Specification of Timing Constraints for Controller Synthesis". Integrated Systems Laboratory, Swiss Federal Institute of Technology Zurich, July 1990. (BDS Volume 1)

16. Pierre Paulin, Cliff Liem, Trevor May, John Knight, "BDS: The BNR Data Structure for DSP Synthesis and Code Generation". Synthesis and Design of Embedded Systems, BNR, April 7, 1992. (BDS Volume 2)
17. Stephan H. Woloszyn, "VOP To BDS Front end". Synthesis, BNR, October 14, 1992. (BDS Volume 3)
18. Pierre. G. Paulin, "High-level synthesis of digital circuits using global scheduling and binding algorithms", Ph. D. thesis, Department of Electronics, Carleton University, January 1988, pp.38-49.
19. Adrian Nye, "Xlib Programming Manual", *X Window System*. O'Reilly & Associates, Inc., January 1995. Adrian Nye, "Xlib Reference Manual", *X Window System*. O'Reilly & Associates, Inc., June 1993. Adrian Nye, Tim O'Reilly, "X Toolkit Intrinsics Programming Manual (Motif Edition)", *X Window System*. O'Reilly & Associates, Inc., February 1995, pp.89,138-148,187.
20. Emil Girczyc, "Automatic generation of microsequenced data paths to realize ada circuit descriptions", Ph.D. thesis, Department of Electronics, Carleton University, July 1984, pp. 72-76

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved