

AD-A178 648

AN IMPLEMENTATION OF A LANGUAGE ANALYZER FOR THE VERY
HIGH SPEED INTEGRAT. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.

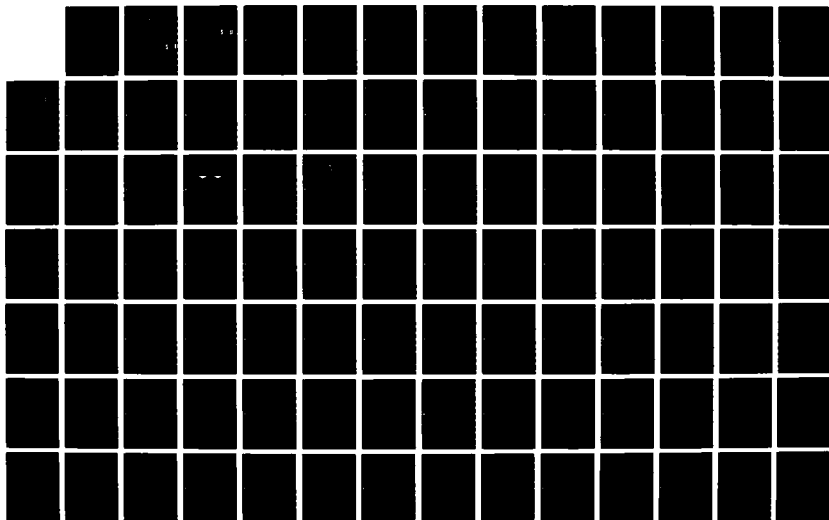
1/2

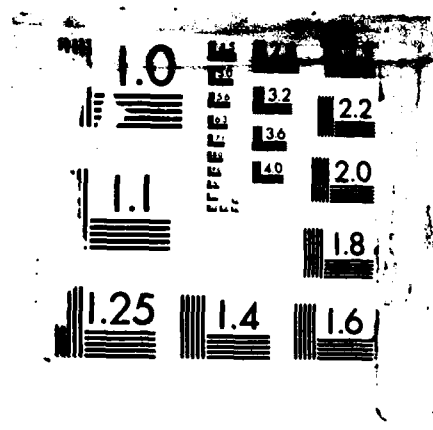
UNCLASSIFIED

D J FRAUENFELDER DEC 86 AFIT/GCE/NA/86D-1

F/G 9/2

NL





MIC

NA

AD-A178 648



AN IMPLEMENTATION OF A LANGUAGE ANALYZER
FOR THE VERY HIGH SPEED INTEGRATED CIRCUIT
HARDWARE DESCRIPTION LANGUAGE

THESIS

Deborah J. Frauenfelder
Captain, USAF

AFIT/GCE/MA/86D-1

DTIC
ELECTE
APR 03 1987
S
D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/MA/86

DTIC
ELECTE
APR 03 1987
S D

AN IMPLEMENTATION OF A LANGUAGE ANALYZER
FOR THE VERY HIGH SPEED INTEGRATED CIRCUIT
HARDWARE DESCRIPTION LANGUAGE

THESIS

Deborah J. Frauenfelder
Captain, USAF

AFIT/GCE/MA/86D-1

Approved for public release; distribution unlimited.

**AN IMPLEMENTATION OF A LANGUAGE ANALYZER
FOR THE VERY HIGH SPEED INTEGRATED CIRCUIT
HARDWARE DESCRIPTION LANGUAGE**

THESIS

**Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering**

**Deborah J. Frauenfelder, B.S.C.S.
Captain, USAF**

December 1986

Approved for public release; distribution unlimited.

Acknowledgments

Thesis research is normally an individual effort; therefore, I was privileged to have worked with a team of researchers whose goal was to create the prototype AFIT VHDL Environment. To my fellow researchers and comrades in arms, I would like to express my deepest appreciation for all assistance rendered during the course of this project. A special thank-you is extended to my thesis readers, Lt Col Harold Carter and Capt James Howatt, for their editorial comments, and to my advisor, Lt Col Gross, for his guidance throughout this project.

Deborah J. Frauenfelder

This report is for open literature. Distribution Statement A is correct.
Per Lt. Col. Richard R. Gross, AFIT/ENG



Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

	Page
Acknowledgments.....	iv
List of Figures.....	vii
List of Tables.....	ix
Abstract.....	x
I. Introduction.....	1.1
Statement of the Problem.....	1.1
Background	1.1
Scope.....	1.5
Research Approach.....	1.7
Maximum Expected Gain.....	1.10
Sequence of the Presentation.....	1.11
II. Survey of Previous Research.....	2.1
Overview.....	2.1
Languages.....	2.2
VHSIC Hardware Description Languages (VHDL).....	2.3
A Design Data Structure (DDS).....	2.6
Summary.....	2.12
III. System Design.....	3.1
Overview.....	3.1
System Requirements.....	3.1
System Organization.....	3.4
Incremental System Implementation.....	3.10
Intermediate Form.....	3.14
Summary.....	3.19
IV. Detailed Design.....	4.1
Overview.....	4.1
Basic Methodology.....	4.2
Design Work.....	4.4
Major Design Decisions.....	4.12
Language Analyzer Detailed Design.....	4.15
Summary.....	4.21
V. Analysis.....	5.1
Overview.....	5.1
Test Requirements.....	5.1
Method of Evaluation.....	5.4

	Page
Evaluation of the VHDL Analyzer.....	5.7
Summary.....	5.14
VI. Conclusions.....	6.1
Principal Conclusions.....	6.1
Suggestions for Future Work.....	6.5
Summary.....	6.11
Appendix A: Deviations from the VHDL Language Reference Manual	A.1
Appendix B: VHDL Intermediate Access (VIA).....	B.1
Overview.....	B.1
Overview of the VIA File Structure.....	B.1
The Component Structure.....	B.2
The VIATABLE Structure.....	B.4
Detailed Record Definitions.....	B.7
Appendix C: Example Test Data.....	C.1
Bibliography.....	BIB.1
Vita.....	VITA.1

List of Figures

Figure	Page
1.1 AFIT VHDL Environment.....	1.3
1.2 Pre-Prototype Syntax Analyzer Design.....	1.4
2.1 VHDL Design Structure.....	2.5
2.2 Directed Acyclic Graph	2.7
2.3 Hierarchical Tree of a Model.....	2.8
2.4 Inter-Subspace Relationships.....	2.9
2.5 Dataflow's One-to-Many Relationship.....	2.10
2.6 Inter-Model Binding.....	2.11
3.1 VHDL Language Analyzer Design.....	3.6
3.2 Parser Generation.....	3.7
3.3 Lexical Analyzer Generation.....	3.8
3.4 Symbol Table.....	3.9
3.5 VHDL Represented in VIA.....	3.18
4.1 Subset 1 -- the Design Entity Shells.....	4.6
4.2 The VHDL Configuration in VIA.....	4.8
4.3 An Example Parser Production for YACC.....	4.11
4.4 High-level Dataflow Diagram of the Language Analyzer.....	4.16
4.5 Attribute Abstract Data Type.....	4.20
4.6 Group Abstract Data Type.....	4.21
5.1 Procedure Test Case.....	5.8
B.1 VIA Record Hierarchy.....	B.3
B.2 VIATABLE Structure.....	B.5
B.3 component Record.....	B.8
B.4 dataflow_link Record.....	B.11

Figure	Page
B.5 dataflow_model Record.....	B.13
B.6 dataflow_net Record.....	B.16
B.7 dataflow_pin Record.....	B.18
B.8 operational_binding Record.....	B.20
B.9 package Record.....	B.22
B.10 single_carrier Record.....	B.24
B.11 single_module Record.....	B.26
B.12 single_node Record.....	B.28
B.13 single_point Record.....	B.30
B.14 single_range Record.....	B.33
B.15 single_value Record.....	B.36
B.16 structural_link Record.....	B.38
B.17 structural_model Record.....	B.41
B.18 structural_net Record.....	B.44
B.19 structural_pin Record.....	B.46
B.20 timing_link Record.....	B.48
B.21 timing_model Record.....	B.50
B.22 undefined Record.....	B.53
B.23 viatable Record.....	B.55
C.1 Test Case 1.....	C.2
C.2 Test Case 2.....	C.3
C.3 Test Case 3.....	C.4
C.4 Test Case 4.....	C.5
C.5 Test Case 5.....	C.6
C.6 Test Case 6.....	C.7

List of Tables

Table	Page
3.1 VHDL Subsets and Capabilities.....	3.15
5.1 Performance Test Raw Data.....	5.11
5.2 Execution Time Means and Standard Deviations.....	5.12

Abstract

This thesis describes the incremental approach used to develop the first known C-based, UNIX-supported translator/analyzer for the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). This research consisted of defining a VHDL Intermediate Access (VIA) format as a translation target, dividing VHDL into manageable segments, describing VHDL-to-VIA relationships, designing software modules to create those relationships, and evaluating the functional and performance characteristics of the analyzer. The intermediate form, VIA, was based upon the Design Data Structure (DDS) developed by Alice Parker and David Knapp.

Three of the nine VHDL language subsets identified were implemented in the language analyzer. In increments, these subsets were manually translated into specific examples of an enhanced version of DDS represented in a pile file format (VIA). These examples were then used as specifications for designing program modules to automatically translate VHDL code into VIA. After the program modules were written, these same examples were used as formal functional test specifications.

I. Introduction

Statement of the Problem

Key members of the microelectronics design community, such as the Institute of Electrical and Electronics Engineers (IEEE) and the Air Force, are on the verge of approving a standard hardware description language, called Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). However, this approval is based solely on a theoretical evaluation of the language, rather than on practical experience gained from integrating VHDL with automated VHSIC design tools.

Background

The Air Force recently forecasted 39 technological goals for the next 25 years. One of the 39 goals is unified life cycle engineering, a "unified, automated design methodology treating performance, manufacturability, and supportability concurrently in computer and design systems" (Kulp, 1986). One step toward the realization of unified life cycle engineering is a hardware description language, such as VHDL, which can both model hardware performance and document hardware design. A hardware description language is a computer language which is used by engineers to describe and to model very high speed integrated circuits and other systems.

Although many hardware description languages exist for describing VHSIC systems, no language has been accepted as an industry-wide standard.

Recognizing this fact, the IEEE selected VHDL Version 7.2 as the basis for a draft hardware description language standard. VHDL contains rules for specifying system requirements, system designs, system components, component behaviors, and multiple component interactions.

Some revisions to the VHDL language were anticipated before the international community approved the standard (AFWAL, 1986). The potential areas for revision were identified and evaluated during 1986. The draft IEEE VHDL Reference Manual (CAD Language Systems, 1986) was released in June 1986. The proposed draft standard was reviewed by the IEEE community during the later half of 1986, and the final standard is scheduled for release in early 1987 (AFWAL, 1986).

To gain the experience with VHDL needed to refine the standard prior to adoption, it seemed necessary to provide the Air Force with some VHDL support tools prior to the delivery of the official contracted VHDL environment. Consequently, in 1985, an Air Force Institute of Technology (AFIT) faculty member proposed that a prototype VHDL support environment be developed by AFIT students. The proposed AFIT VHDL Environment (AVE), depicted in Figure 1.1, consists of six high level components: a VHDL analyzer, a VHDL code checker, a software simulator, a simulator generator, a hardware simulator engine, and a VHDL microcode compiler (Carter, 1985).

The VHDL analyzer is a computer program which translates VHDL programs describing circuits into an intermediate form which is eventually processed by other tools in the environment. A pre-prototype VHDL analyzer

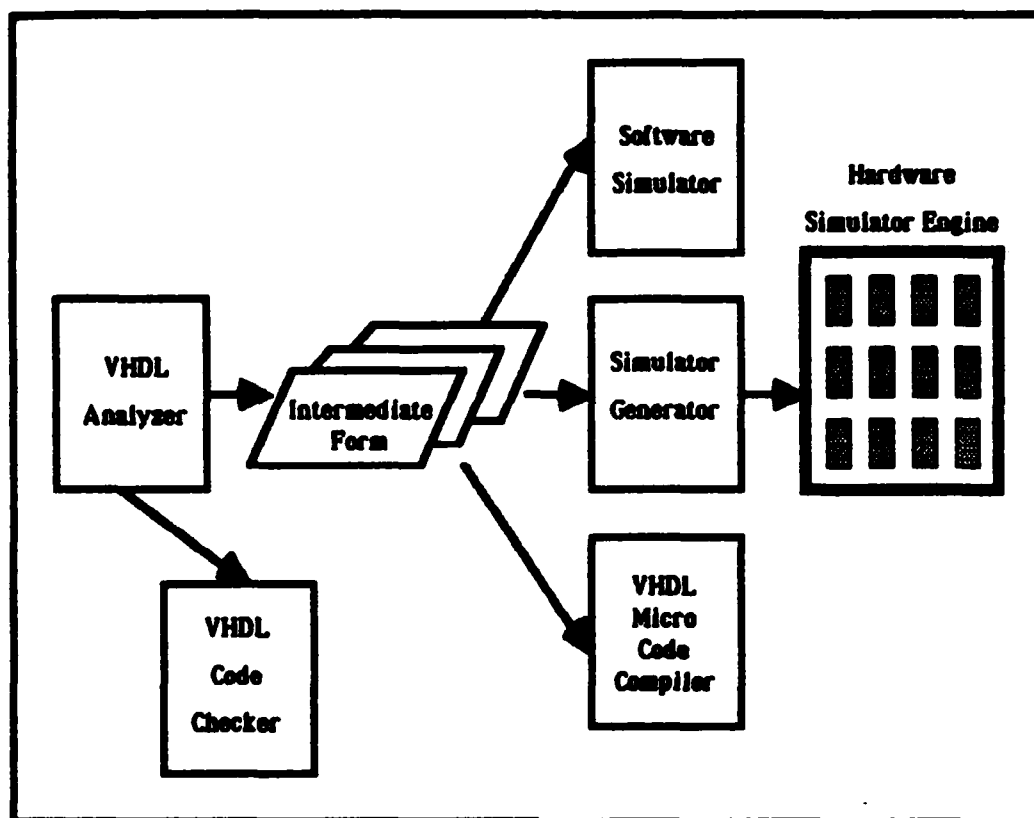


Figure 1.1: AFIT VHDL Environment.

(see Figure 1.2) was developed by the author for a class project. This pre-prototype analyzer successfully recognized the syntax of the entire VHDL language.

A code checker is a computer program which analyzes the VHDL program to determine logic errors, circuit timing errors, and potential optimization areas. The VHDL analyzer initiates the VHDL code checker when a user requests the action.

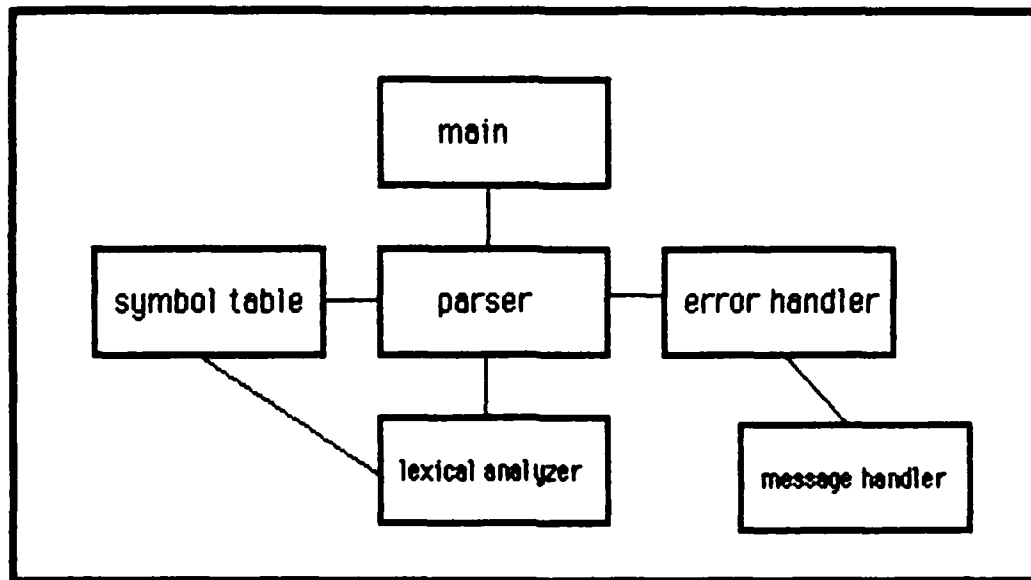


Figure 1.2: Pre-Prototype Syntax Analyzer Design

A software simulator is a computer program which models the behavior of the circuit. The circuit represented in the intermediate files describes the signals, the data, and the control sequencing used by the software simulator.

A simulator generator is a computer program which controls the operation of a different computer (the hardware simulation engine) based upon information contained in the intermediate files. The simulator generator and the software simulator conceptually perform the same control tasks, but the simulator generator also controls the parallel tasking of the hardware simulation engine.

A hardware simulator engine is a computer hardware system designed to model the behavior of circuits. The hardware simulator engine provides the same information the software simulator provides. The hardware simulator

engine is expected to model the circuit up to an order of magnitude faster than the software simulator.

A microcode compiler is a computer program which reads procedure inputs and the intermediate code produced by the VHDL analyzer to generate tables of information in the form of computer code. This computer code generates the microcode for the computer described by the VHDL code.

Scope

The primary goal of this research endeavor was to implement a prototype VHDL language analyzer which provided the functional capabilities necessary to perform as a front-end processor for a multiple tool design environment. To achieve this goal, the following subtasks were established:

1. Select an interface structure for the VHDL Design Environment.
2. Identify the VHDL constructs to implement.
3. Identify the logical relationship between the VHDL constructs and the interface structure.
4. Design modules to generate the intermediate structure.
5. Create the package STANDARD specified in the VHDL Language Reference Manual (Intermetrics, 1985a: B-7 to B-9).
6. Test and evaluate the language analyzer.

This research was considered to be finished when:

1. the analyzer's functional capabilities and requirements were defined;
2. the language analyzer was designed, coded, and tested;

3. the functional requirements were compared to the language analyzer's performance; and
4. the results were reported in this thesis.

Certain less important areas were not addressed by this study because of resource constraints:

1. Optimization of the intermediate structure generated by the language analyzer was excluded from the scope of the research because the VHDL language will change next year. As stated earlier, in 1986 the IEEE was refining VHDL to become the industry standard hardware description language. During the standardization process, the IEEE changed VHDL's grammar. The constructs of the language subject to change were not known at the start of this research endeavor. After the language has been approved by the IEEE, the VHDL language analyzer should be modified to reflect the changes. Because of these facts, research time considered for allocation to optimization would best be used elsewhere.
2. Creation of a design library and a design library manager, with the exception of the STANDARD package mentioned earlier, was excluded from the scope of the research due to the complexity of the task. The package STANDARD is essential for the AFIT VHDL Environment because VHDL semantics assume the existence of the package STANDARD. Even in this case, the design library manager function can be provided by the UNIX support environment.

Research Approach

The following approach was used to address research subtasks in this project.

1. Select an interface structure for the VHDL Design Environment.

Selecting a structure for the intermediate files is the process of analyzing the VHDL language to determine a method for representing the content of a VHDL source program. Two candidate methods are Intermediate VHDL Access Notation (IVAN) as presented in the VHDL Design Library Specification (Texas Instruments, 1984:3-1 to 3-21) and Design Data Structure (DDS) as presented by Knapp and Parker of the University of Southern California (Knapp and Parker, 1984:9-27). Both methods are well documented, and either would have enabled the project to continue with minimum design time. An alternative to using an existing structure was to design a structure explicitly for VHDL. This alternative would have required at least six months of research time. The added research time would have delayed the production of the intermediate files, which were required for the parallel development projects. Therefore, one of the documented structures was selected based on a comparison of their respective capabilities to represent the semantic content of VHDL. Selecting the structure which best represents VHDL semantics insured the tools under development had access to efficient circuit descriptions.

2. Identify the VHDL constructs to implement. Identifying the VHDL constructs to implement is the process of classifying the language rules into

six or more subsets of rules. To expedite implementation of some capability for the parallel projects to use, the rule subsets were ranked in the order of expected implementation complexity. As each rule set was added to the analyzer, the analyzer's capabilities expanded. Computer code was designed and tested to validate the expanded capabilities. The completed analyzer will eventually contain all the rule subsets.

An alternative to creating subsets for the language would have been to design each phase of the analyzer separately. For example, the lexical analyzer would have been completely designed and tested; then, the parser would have been designed and tested. This alternative method would mean the intermediate files could have been generated only after the entire project was complete. Traditionally, the alternative method has been used, but in this case four parallel development projects (see Figure 1.1) required the intermediate files. The method of selecting subsets of the VHDL language allowed earlier access to the intermediate files, enabling the parallel projects to progress.

3. Identify the logical relationship between the VHDL constructs and the interface structure. Identifying the logical relationship between the VHDL constructs and the interface structure is the process of determining how the semantics of the language are explicitly represented in the intermediate form. The language subsets (identified in the preceding task) were iteratively decomposed to provide examples of VHDL source code and the intermediate form. The examples served as a guide for designing the modules in step 4; they formally specified the interface for the parallel projects; and,

they formed the test cases used for validation in step 6.

An alternative to the iterative approach was to decompose the entire language before preceding to the next step. The approach would reduce the risk of errors due to an incomplete understanding of the complex logical relationships, but in this case four parallel development projects (see Figure 1.1) required the intermediate files. The iterative method allowed earlier access to the intermediate files, enabling the parallel projects to progress.

4. Design modules to generate the intermediate files. Designing modules to generate the intermediate structure is an iterative process of determining the actions required for generating the intermediate files. The example code created in step 4 was used to create tables and narrative descriptions specifying the required actions. Once the actions were identified, programs were written to simulate the actions. The design process is a fundamental step for any software development project. In this case, the design process not only documented the detailed design, but also, formed the basis for the maintenance manual.

5. Create the package STANDARD specified in the VHDL Language Reference Manual (Intermetrics, 1985a: B-7 to B-9). Creating the predefined attributes, types, and subtypes of the package STANDARD is the process of writing VHDL source code to build the primitive environment for VHDL as defined in the language reference manual (LRM). This environment was not essential for the creation of the language analyzer, but it was essential for the AFIT VHDL Environment. Without the primitive types and attributes

specified in the LRM, each person who wrote a VHDL program would need to write the code for the primitive types and attributes he used. Additionally, the semantics of the VHDL language assume the existence of the package STANDARD. Step 6 used the VHDL source code for the package STANDARD.

6. Test and evaluate the language analyzer. Testing and evaluating the language analyzer is the process of executing the VHDL language analyzer, checking the results against the predicted results, verifying the expected output, and determining run time performance. All VHDL source code written for steps 3 and 5 was applied as test data. Additionally, VHDL written by others, such as the AFIT VHDL Beta Test Team and AFIT's hardware architecture classes, was applied as test data. The raw data was gathered from the each test and statistically analyzed to determine the run time performance of the language analyzer. The data collection techniques and the specific metrics used in analyzing the performance are specified in Chapter 5.

Maximum Expected Gain

The maximum expected gain for this research endeavor was the development of the first known UNIX-resident VHDL language analyzer which successfully recognized the entire VHDL language and produced a useful intermediate form. The language analyzer was targeted to emphasize function, that is, to handle the entire language with satisfactory performance as a secondary goal. Satisfactory performance was defined as averaging less than 3 minutes CPU time for analyzing a single 1000-word VHDL source code file.

Sequence of the Presentation.

This thesis contains six chapters with three supporting appendices. After the introduction in Chapter 1, Chapter 2 presents a survey of previous research on languages, on VHDL, and on an intermediate language abstraction called Design Data Structure (DDS). Chapter 3 describes the AFIT VHDL Language Analyzer's system design, specifies the system requirements, and describes the intermediate form selected. In turn, Chapter 4 describes the detailed design of seven incremental designs which as a composite create the language analyzer. The test methodology and analytical results are presented in Chapter 5. Then Chapter 6 summarizes the conclusions and recommends future research endeavors.

Three appendices were written to support the body of this thesis. Appendix A contains a list of VHDL language requirements which were specified in the VHDL Language Reference Manual (Intermetrics, 1985a), but were not implemented in the prototype language analyzer. Appendix B contains the complete VHDL Intermediate Access (VIA) format specification. Appendix C contains selected examples of test data which were used to validate the prototype language analyzer.

II. Survey of Previous Research

OVERVIEW.

The development of electronic systems is a complicated process which encompasses many diverse tasks. These tasks include, but are not limited to, specifying system performance requirements; specifying system functional requirements; functional design; logic design; simulation and modeling; mask placement and routing; fabrication; and testing. As with any research endeavor, the development of electronic systems is inevitably subject to change (Dewey and Gadiant, 1986: 12) due to either fluctuating user requirements or advances in technology. Regardless of the cause, the effect is the same -- increased cost and increased time expenditures.

In recent years, to reduce cost and time (Dewey and Gadiant, 1986: 13), many hours of research have been dedicated to the design and development of computer-aided design environments specifically tailored for electronic system design. From these research hours, many Hardware Design Languages (HDL's), such as VHDL, have emerged. To support these HDL's, language environments, language analyzers, and intermediate language representations have evolved. In the following sections, after a general background survey on languages, VHDL is briefly described, and an intermediate language abstraction, called Design Data Structure (DDS), is discussed.

Languages.

Many languages currently exist which support the various electronic system design tasks (Aylor, Waxman, and Scarratt, 1986: 17). Although some traditional general-purpose programming languages are still used for designing electronic systems (Katzenelson and Weitz, 1986: 371), such design programs are time-consuming to write and tend to be application-specific, offering minimal, if any, reuse capability. Katzenelson and Weitz showed the importance of applying data abstraction to the design of electronic systems to reduce time and increase the generic reuse capability. They also showed how data abstraction can lead to specification clarity and avoid unnecessary program detail.

Some high level general-purpose programming languages support data abstraction and thereby allow structural and procedural descriptions of an electronic component to be developed. Since most high level language statements are sequentially executed, a general-purpose language programmer developing a simulation model of an electronic component has full responsibility to encode control flow to simulate the concurrent effects of the electronic component. This extra code tends to proliferate unnecessary program detail, which Katzenelson partially avoids by using a high level language with data abstraction.

Due to the insufficiencies of traditional languages a new class of languages, Hardware Description Languages (HDL's), emerged in the 1960's. Initially many of the HDL's, such as IDL (Interactive Design Language) or CDL

(Computer Design Language), were designed to handle a relatively small class of electronic components. IDL was developed by International Business Machines (IBM) Corporation to design programmable logic arrays (PLA's); and CDL was developed to teach digital logic design. More recently, HDL's have matured to describe a more general class of electronic components (Aylor, Waxman, and Scarratt, 1986: 22).

In their study of HDL's, Aylor, Waxman, and Scarratt provided 12 factors for evaluating modern HDL's. Among these factors were the "range of hardware to be described ..." and "... language extensibility". The authors indicated that a HDL should "support the complete description and design process" from the most abstract to the most detailed description of a hardware component. Finally, the language should be technology independent and capable of growing as technology advances (Aylor, Waxman, and Scarratt, 1986: 18-22).

Dewey and Gadiant supported the choice of these criteria, stating that simultaneously specifying an interface and documenting a design lead to the assurance of having a system's properties and characteristics accurately reflected in the completed design (Dewey and Gadiant, 1986: 13). In fact, as the first contract monitor for VHDL, Dewey embedded these criteria into the specification for VHDL.

VHSIC Hardware Description Language (VHDL).

In response to such perceptions as these, the requirements for VHDL were established in 1981 by the United States Air Force (as agent for the

Department of Defense) in an attempt to "reduce IC design time and effectively insert VHSIC technology into military systems" (Dewey and Gadiant, 1986: 12). In the early stages of VHDL design, an extensive analysis of existing languages and their environments was performed to extract the major advantages of each (Aylor, Waxman, and Scarratt, 1986: 17). By 1983 the requirements were firmly established and the design of the VHDL language specification began (Dewey and Gadiant, 1986: 12).

The language which most greatly influenced the grammar of VHDL was Ada, a high level general-purpose programming language which was also sponsored by the Department of Defense. Like Ada, VHDL has both the traditional procedural and functional capabilities as well as the more modern capabilities associated with data abstraction, such as type definition, subtype definition, operator overloading, and packaging. Yet, unlike Ada, VHDL is a hardware description language. The difference is fundamental: a hardware component is a physical unit which operates on all inputs concurrently to produce an output. To describe hardware accurately, three *design entities* were incorporated into VHDL: *interfaces*, *architectures* and *configurations*. An interface specifies the physical input/output ports available on a semiconductor chip; an architecture (the principal unit used to describe a chip) specifies the chip's internal operation; and a configuration specifies how the chip's ports are connected to the external world, such as board connections. Functional subdivision into these three design entities provides the capability to describe differing chip architectures without changing the ports or the port connections; it also permits hierarchically defined architectures. An abstract example of the design entity relationships is depicted in Figure 2.1.

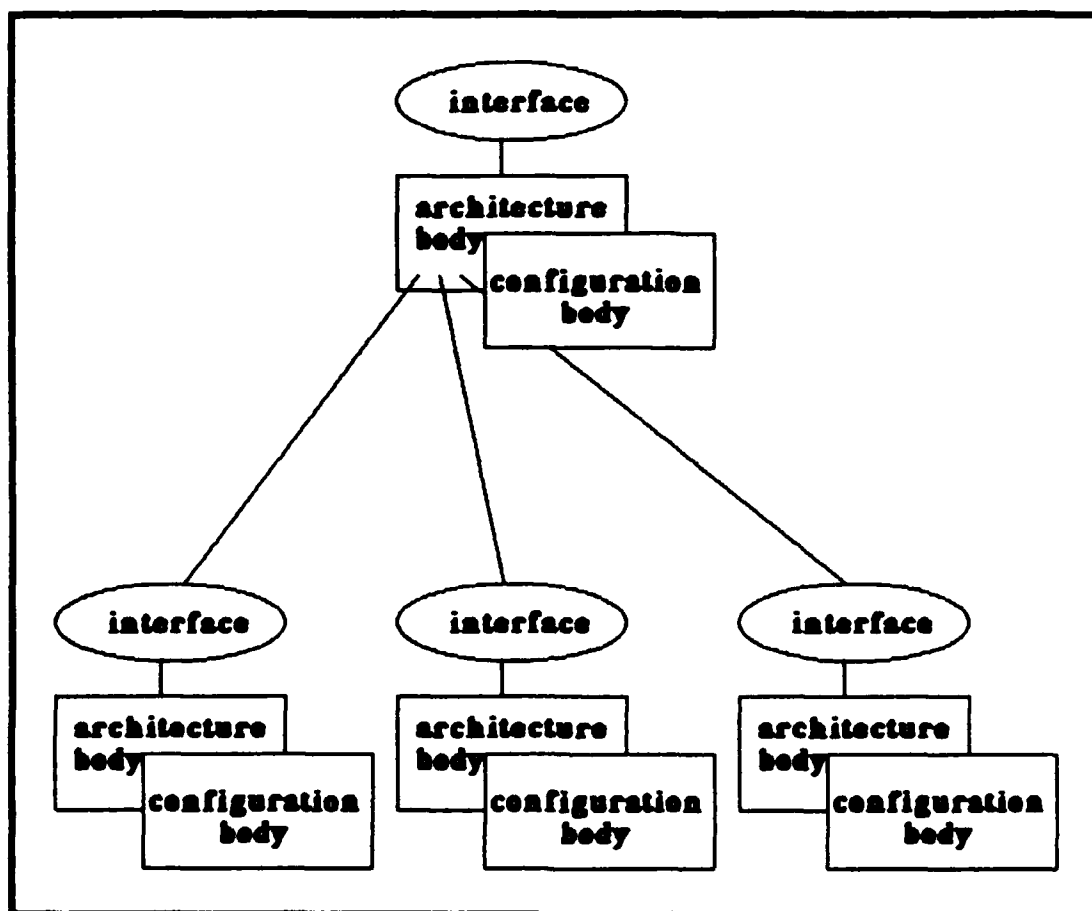


Figure 2.1: VHDL Design Structure.

A chip description can be constructed hierarchically, using either *concurrent* statements or *sequential* statements. Concurrent statements are statements which simultaneously execute, while sequential statements execute in the order which they are encountered. The architecture which contains only concurrent statements is called a *structural* description, while the architecture which contains only sequential statements is called a *behavioral* description. An architecture can contain both concurrent and sequential statements. Concurrent statements primarily operate on signals, while sequential statements operate on variables. A signal is similar to a

variable, except that signals include the notion of time.

Although the preceding discussion of VHDL was necessarily cursory, the VHDL Language Reference Manual (Intermetrics, 1985a) contains a complete description of the language. In the following section, a general background for an intermediate VHDL data structure, called Design Data Structure (DDS), is discussed.

A Design Data Structure (DDS).

DDS (Afsarmanesh and others, 1985: 14-44) was developed by Knapp and Parker in 1984 at the University of Southern California. DDS is a method of representing four abstract views of a hardware model: *dataflow*, *timing*, *structure* and *physical* (Knapp and Parker, 1984: 10-13). The dataflow view represents functions and the values associated with a functional transformation. The timing view represents the range of time under which the transformations occur. The structural view represents a schematic diagram with its components and interconnections. The physical view¹, in contrast to the structural view, represents the actual size and placement of a component and the size and placement of wires.

The four abstract views are not independent. When considered together with their dependencies, the views form a directed acyclic graph. Figure 2.2

1. Within the scope of the prototype AFIT VHDL Environment (AVE) project, the physical view of DDS will not be used. However, the concepts of the physical view will be preserved for eventual integration of the AVE into a unified AFIT VLSI design environment which uses one central database created using the concepts of DDS.

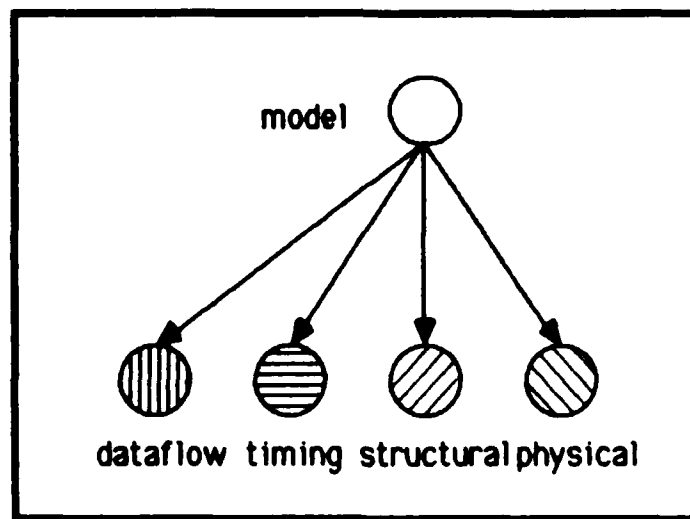


Figure 2.2: Directed Acyclic Graph.

presents a simple directed acyclic graph which shows the four views of a hardware model. In graph theory, the circles in Figure 2.2 would be called vertices or nodes. Yet, within the scope of DDS, the terms *vertex* and *node* are not synonymous. The circles are called circles or vertices; the term *node* is a name for a type or a class of vertices. Furthermore, in graph theory, the arrows in Figure 2.2 would be called *arcs* or *relations*. Within the scope of DDS, the arrows are called *arrows*, *relationships* or *bindings*. The pattern within a circle represents the type of vertex. A clear circle always has a type name associated with the vertex. For instance, the circle with vertical bars in Figure 2.2 is a vertex of type *dataflow*. In Figure 2.3, the clear circle at the top of the figure is a model vertex, the circle with the horizontal lines is a timing vertex, and the timing vertex is bound to a range vertex. The term *subspace* means the set of vertices and bindings subordinate to a dataflow, timing, structural, or physical vertex. In Figure 2.2, the dataflow subspace consists of one vertex, the dataflow vertex. In Figure 2.3, the dataflow

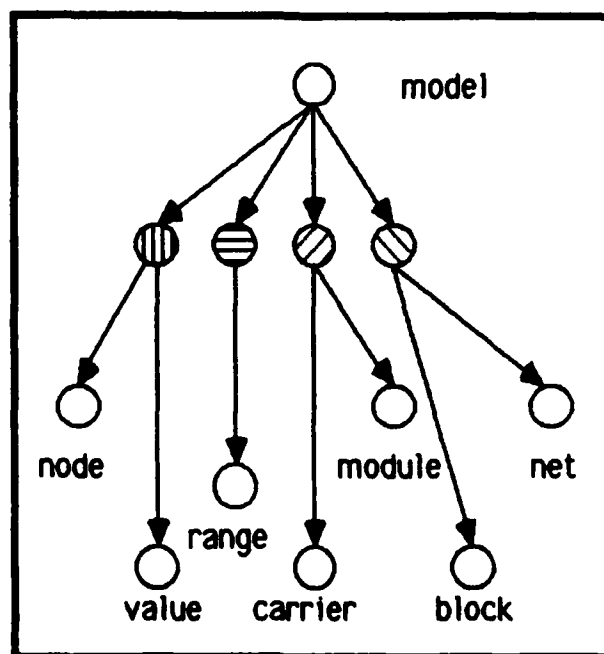


Figure 2.3: Hierarchical Tree of a Model.

subspace consists of three vertices and two bindings.

The primary graph for representing any hardware model is the tree depicted in Figure 2.3. The tree represents the hierarchy of the model, with the first level of the hierarchy representing the dataflow, timing, structural and physical subspaces. Each subspace is further decomposed into one or more components. The dataflow subspace consists of nodes and values. The node vertices represent functional transformations. The value vertices represent the results of functional transformations. The timing subspace consists of ranges of time. The range vertices represent an interval of time required to control the flow of the functional transformations. The structural subspace consists of carriers and modules. The carrier vertices represent interconnecting lines on a schematic diagram over which the values in the

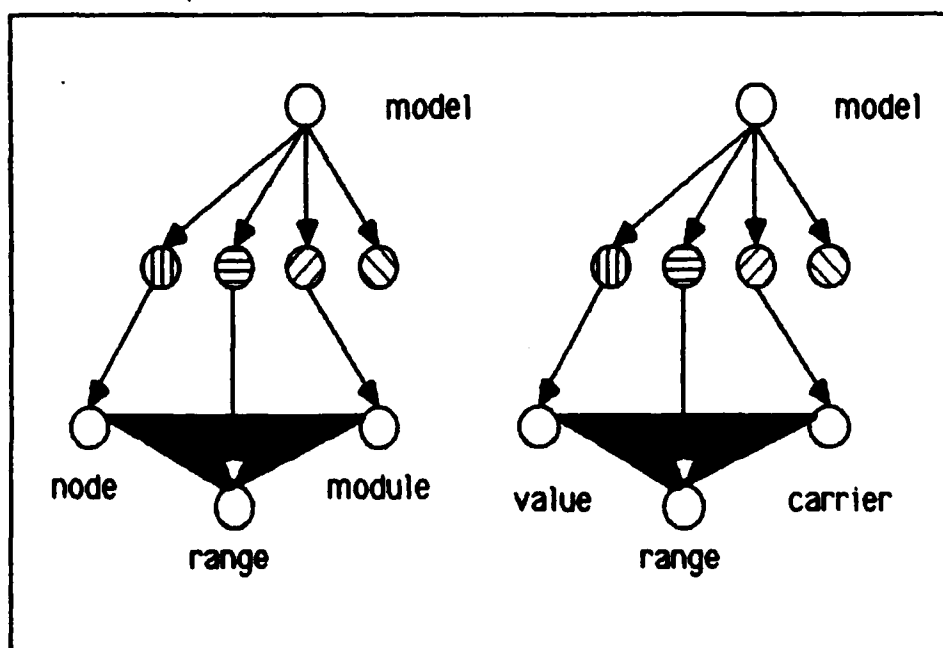


Figure 2.4: Inter-Subspace Relationships.

dataflow subspace are carried. The module vertices represent the components on a schematic diagram, within which functional transformations occur. The physical subspace consists of blocks and nets. The block vertices represent the physical features of a design mask layout. The net vertices represent interconnecting wires in the design layout. Blocks and nets are related to modules and carriers, but the blocks and nets have attributes such as size, orientation, layer, and technology.

As the preceding discussion pointed out, the subspaces are interrelated in several ways. The black triangles in Figure 2.4 represent the intersubspace bindings which occur at a lower level in the hierarchy. These intersubspace bindings transform the hierarchical tree into a directed acyclic graph. As Figure 2.4 shows, two distinct inter-subspace bindings exist:

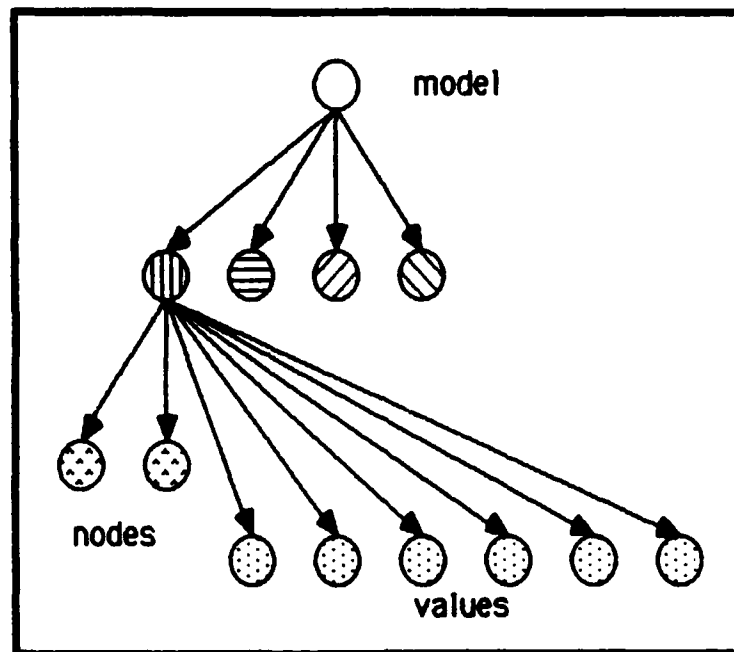


Figure 2.5: Dataflow's One-to-Many Relationships.

carrier-value-range and module-node-range bindings (Knapp and Parker, 1984: 14). Both inter-subspace bindings change with respect to time. For instance, suppose at time t_0 wire A carries a 5 volt charge, and at time t_1 the charge is drained. In this simple example two carrier-value-range bindings are established: A-5- t_0 and A-0- t_1 . Additionally, a binding internal to the time subspace was established: time t_0 occurs before time t_1 .

The root of a subspace has a one-to-many relationship with its subordinate vertices. For instance, Figure 2.5 shows a model whose dataflow subspace has two nodes and six values while the other subspaces have no subordinate vertices. A model such as this could easily represent all that is known about a hardware component at the earliest stage of the design process. For example, perhaps the designer knows the initial conditions for two

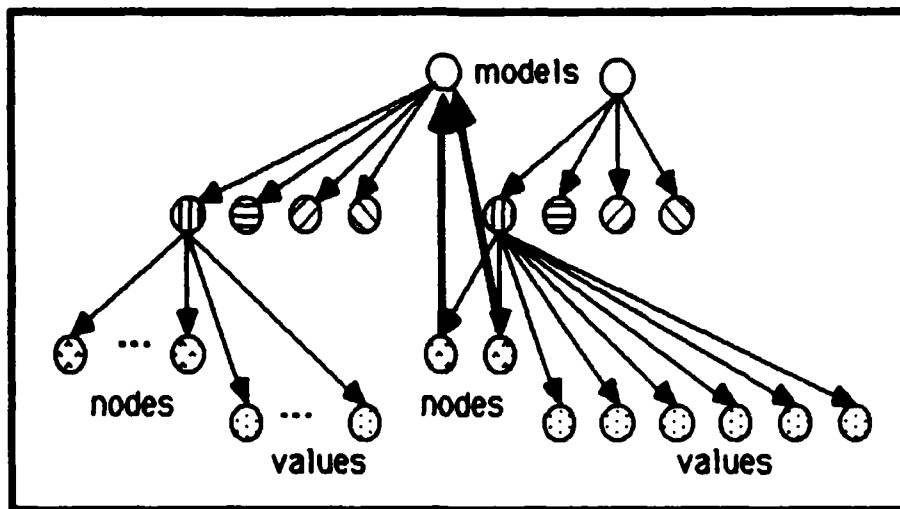


Figure 2.6: Inter-Model Binding.

functions which will be performed. As the design process continues, the designer may discover that the two functions represented by the nodes in Figure 2.5 are actually identical operations which could be represented by a second model. The second model would have a set of subspaces as depicted in Figure 2.6. At this stage of the development, both nodes in the first model point to one copy of the second model. At some point in the design process, the designer would have sufficient information about both models to describe their structure and behavior. At that time, two copies of the second model would be created and inter-model subspace bindings would be created for all subspaces. The inter-model subspace binding essentially reflects shared assets.

Knapp and Parker present a complete list of characteristics of these subspaces and the reader is referred to their work for further detail (Knapp and Parker, 1984: 35-61).

Summary.

Although electronic circuits were first modeled using high level software languages, the need to reduce cost, to reduce time, and to simplify documentation led to the development of VHDL. VHDL has not only the programming power of a general-purpose programming language, but also, three design entities (interfaces, architectures and configurations) which enable modeling an electronic circuit. These entities, coupled with the embedded concepts of signals and concurrent statements, strengthen VHDL. VHDL allows top-down design using architectures independent of the interface and configuration. The hierarchy embedded in VHDL architectures can easily be preserved and represented in the Design Data Structure (DDS) of Knapp and Parker. VHDL and DDS share three important concepts: behavior (or dataflow), time, and structure. In the next chapter, the relationship between VHDL and DDS is defined in terms of an intermediate form called VHDL Intermediate Access (VIA). The complete VIA specification is provided in Appendix B.

III. System Design

Overview.

Designing a system is the process of specifying system requirements, translating those requirements into a functional system organization, determining the external input, and establishing the desired output. Therefore, four topics are discussed in this chapter: the language analyzer system requirements, the language analyzer system organization, the VHDL input subsets translated by progressive implementations of the system, and the intermediate form output, called VHDL Intermediate Access (VIA).

System Requirements.

The environmental, functional, and performance requirements for the language analyzer system are summarized as follows:

1. Embed the language analyzer in the UNIX environment.
2. Support a wide range of VHDL design tools.
3. Analyze the syntax and semantics of VHDL, Version 7.2

(Intermetrics, 1985a).

4. Emphasize user friendliness.
5. Facilitate ease of maintenance.
6. Process a 1000-line input file within three minutes of CPU time.
7. Analyze one input file per execution of the language analyzer.
8. Reduce output file size.

For this project, code optimization was considered a non-important requirement; rather, as discussed in Chapter 1, the emphasis for this project was to produce a functionally correct prototype analyzer. Although good software development techniques, such as structured design, information hiding, and structured code, were applied to this project, the emphasis was on building a functional initial prototype, rather than on performance.

The rationale behind these requirements is explained below.

1. Embed the language analyzer in the UNIX environment. As mentioned in Chapter 1, the language analyzer is the front-end processor for the prototype AVE environment. Since AVE is designed to reside on the UNIX system, the language analyzer by default must execute on the UNIX system.
2. Support a wide range of VHDL design tools. As mentioned in Chapter 1, at the start of this project three AVE tools were identified to interface with the language analyzer. Potentially other design tools will eventually be designed to interface with the language analyzer. With an open-ended set of AVE design tools, the language analyzer must be designed independent of any specific tool and emphasize an interface for a wide range of tools.
3. Analyze the syntax and semantics of VHDL, Version 7.2 (Intermetrics, 1985a). As mentioned in Chapter 1, during 1986 the IEEE was establishing an industry-wide hardware description language based on VHDL, Version 7.2. So the draft IEEE standard was an alternative to using VHDL Version 7.2 as the baseline for this project. Yet, this requirement would have increased the risk associated with a successful completion of the analyzer due to fluctuating baseline requirements. Therefore, VHDL Version 7.2 was identified as the

baseline definition for this project.

4. Emphasize user friendliness. The analyzer is intended to support students in an academic environment. Therefore, the analyzer must be easy to operate, support a meaningful "help" capability, provide clear concise error messages, provide meaningful output, and execute within a reasonable period of time.

5. Facilitate Ease of maintenance. As mentioned in Chapter 1, this project was an incremental development effort. So, by necessity, the analyzer must be easy to modify not only during initial development, but also during the follow-on refinement to incorporate the changes to VHDL generated by the IEEE community.

6. Process a 1000 line input file within three minutes of CPU time. As discussed in Chapter 1, the prototype analyzer emphasizes function with performance as a secondary goal. Nevertheless, a minimal acceptable baseline was arbitrarily established by the author: analysis of 1000 lines of VHDL code within three CPU minutes.

7. Analyze one input file per execution of the language analyzer. A more useful analyzer would process multiple input files; yet, analysis of multiple files would necessitate developing a linker for the associated output files. Since a linker was not defined within the scope of this project, the number of input files was limited to one. However, within that VHDL source file, multiple design entities may be defined because VHDL requires multiple design entity interaction.

8. Reduce output file size. Two conflicting constraints regulate the optimization of the output file size: maximizing readability and minimizing wasted space. Since tool builders read the files generated by the language analyzer, the information in the files must contain sufficient information to identify the contents. Yet, at the same time, unnecessary information should be minimized so reduction of the output file size was established as a requirement.

System Organization.

Two potential design methodologies existed for deriving the system organization: create a system design based upon the aforementioned requirements, or tailor an existing design to meet them. The first method provided the advantages of performing a complete top-down system design. A top-down system design would ensure all system requirements were decomposed and efficiently translated into the end product. Yet, the first method gave the distinct disadvantage of increased design time. As mentioned in Chapter 1, three parallel projects were associated with this project in the creation of the prototype AFIT VHDL Environment (AVE). The success of the prototype AVE depended upon an early prototype language analyzer. Therefore, the second method was used.

In addition to decreased design time, the second method provided the following advantages:

1. Facilitated transfer of technology. Transfer of technology was achieved in three ways. First, the initial design was based upon a C compiler developed by Schreiner and Friedman (Schreiner and Friedman, 1985). Their design was

selected because the design and the code for the program modules were well documented. Second, some design modification decisions were based upon work done by Intermetrics while under contract to the Department of Defense (Intermetrics, 1986c). Intermetrics specified the semantic actions for a VHDL parser written in Ada. These semantic actions were modified for the AFIT VHDL parser. Third, the intermediate form was based upon the Design Data Structure (DDS) developed by D. W. Knapp and A. C. Parker at the University of Southern California (Knapp and Parker, 1984). Using DDS as the underlying structure for VHDL Intermediate Access (VIA) reduced the research time required to specify the intermediate form during the system design phase.

2. Extensive use of computer aided design tools. As stated earlier, the initial design was based upon Schreiner's and Friedman's work (Schreiner and Friedman, 1985). They used two computer-aided design tools: *LEX* and *YACC*. *LEX* is a lexical analyzer generator (Lesk and Schmidt, 1978), and *YACC* is a parser generator (Johnson, 1978). Both *LEX* and *YACC* were available for use while the author was developing the VHDL language analyzer. Both tools facilitate information hiding; reduce development and maintenance time; and generate C code. Therefore, in order to make use of C-based computer-aided design tools, C was selected as the implementation language.

The system organization which evolved is depicted in Figure 3.1. A main driver program calls the parser. The parser checks the grammar calling the lexical analyzer for tokens. When the parser needs information about a literal, the parser calls the symbol table routines. Upon finding an error, the parser calls the message handler routines. The lexical analyzer finds tokens

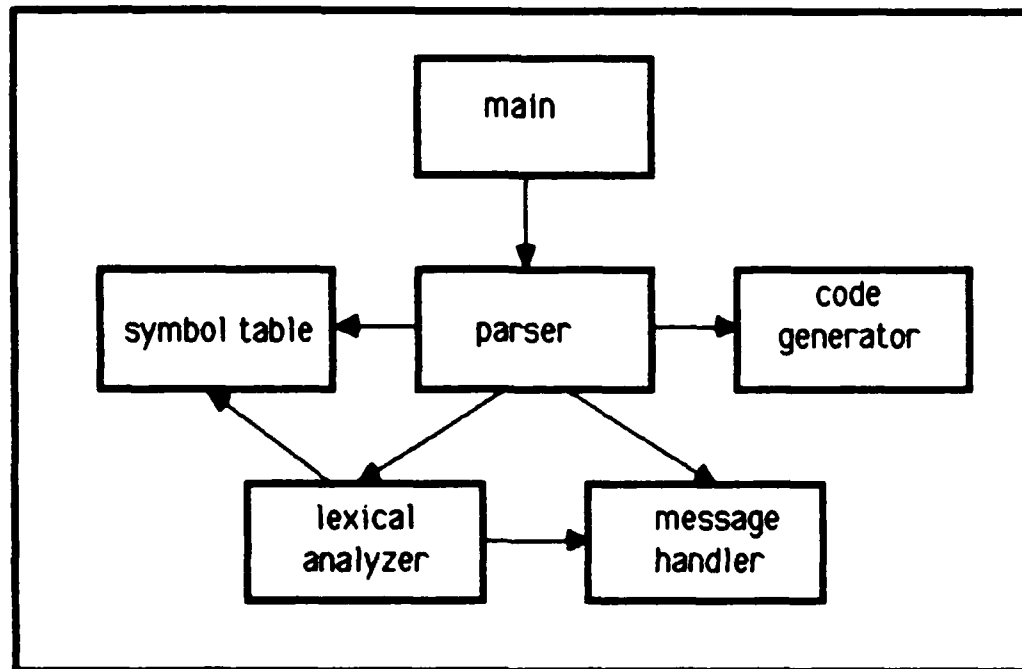


Figure 3.1: VHDL Language Analyzer Design

in the input file and passes the tokens back to the parser. When the lexical analyzer reads a literal, the literal is entered into the symbol table. When the lexical analyzer reads an undefined sequence of characters, the message handler prints an error.

As mentioned earlier, the basic design of the language analyzer was derived from Schreiner's and Friedman's work. Of the six modules depicted in Figure 3.1, two required no tailoring: *main* and *message handler*. The modifications to the other four are explained below:

Parser. Since Schreiner's and Friedman's parser recognized a subset of the C language, it was necessary to modify their parser to recognize VHDL. As mentioned earlier, the parser is generated by YACC (Johnson, 1978) (see

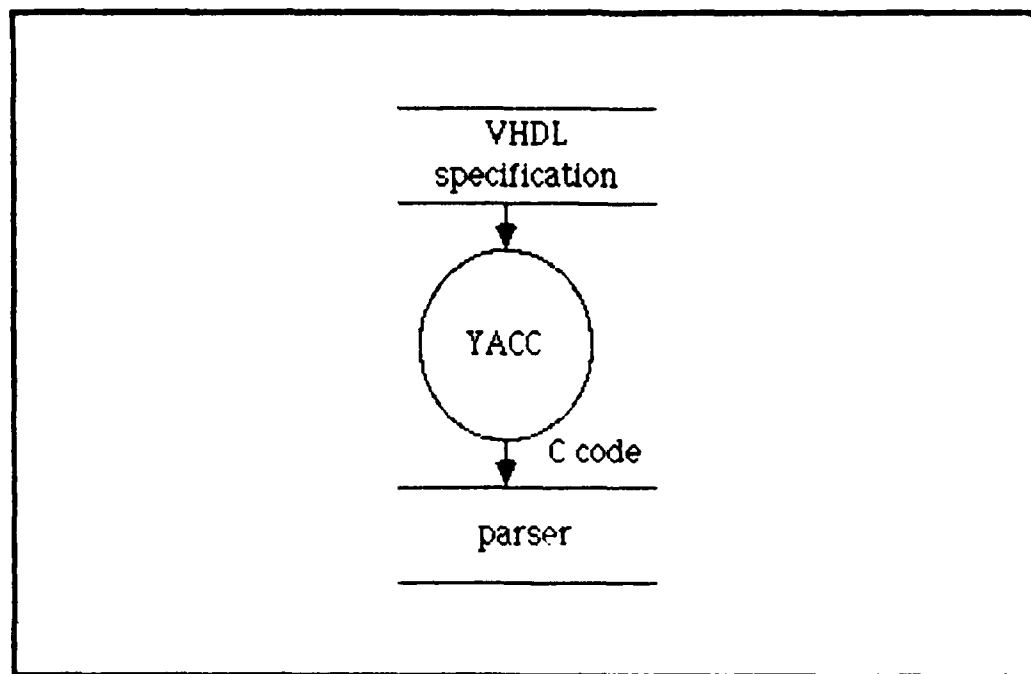


Figure 3.2: Parser Generation.

Figure 3.2). The VHDL specification contains production rules and semantic actions which describe VHDL. These production rules are similar to the grammar rules defined in the VHDL Language Reference Manual (Intermetrics, 1985a: C-1 to C-20). The production rules essentially allow the parser to analyze the syntax of the VHDL source code, while the actions analyze the semantics.

Lexical Analyzer. Since Schreiner's and Friedman's lexical analyzer recognized tokens for the C language, it was necessary to modify their lexical analyzer to recognize VHDL tokens. As mentioned earlier, the lexical analyzer is generated by LEX (Lesk and Schmidt, 1978) (see Figure 3.3). The token specification contains VHDL token definitions and their classifications. The token definitions are based upon the lexical elements defined in the VHDL

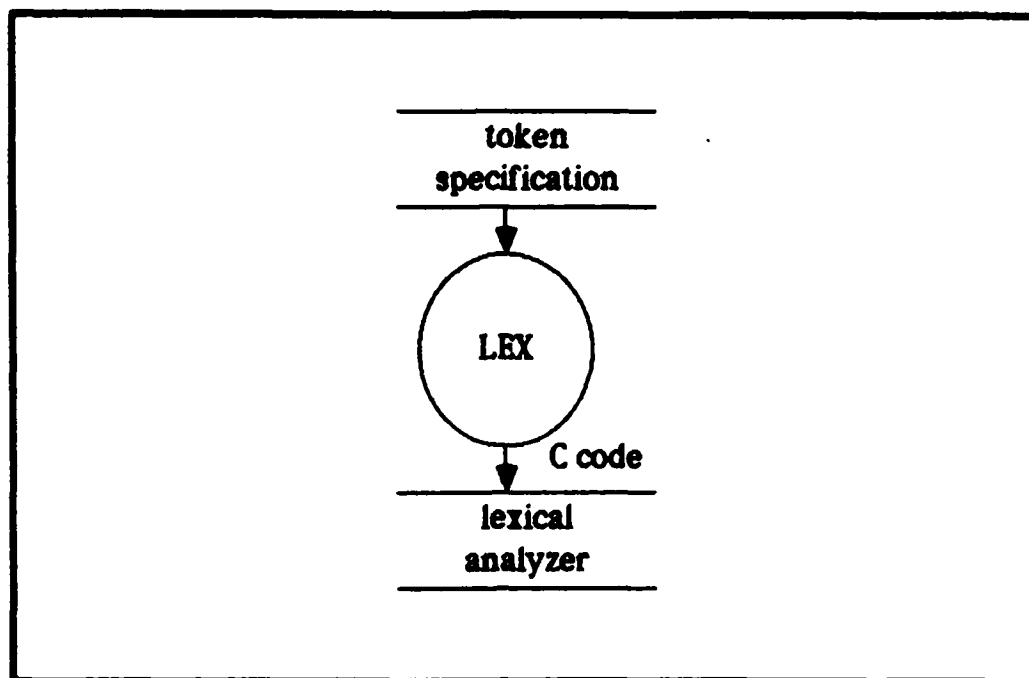


Figure 3.3: Lexical Analyzer Generation.

Language Reference Manual (Intermetrics, 1985a: A-1 to A-11).

Symbol Table Processor. Since Schreiner and Friedman designed their symbol table for the C language, it was necessary to modify their symbol table processor to process VHDL symbols. The symbol table processor is a set of functions which maintains a table of all identifiers and literals. The symbol table is central to the correct operation of the language analyzer. The symbol table processor not only maintains type information, but also maintains scope information for all identifiers/literals. Since the type and scope information required by Schreiner and Friedman was different for VHDL, the symbol table concepts were modified as depicted in Figure 3.4.

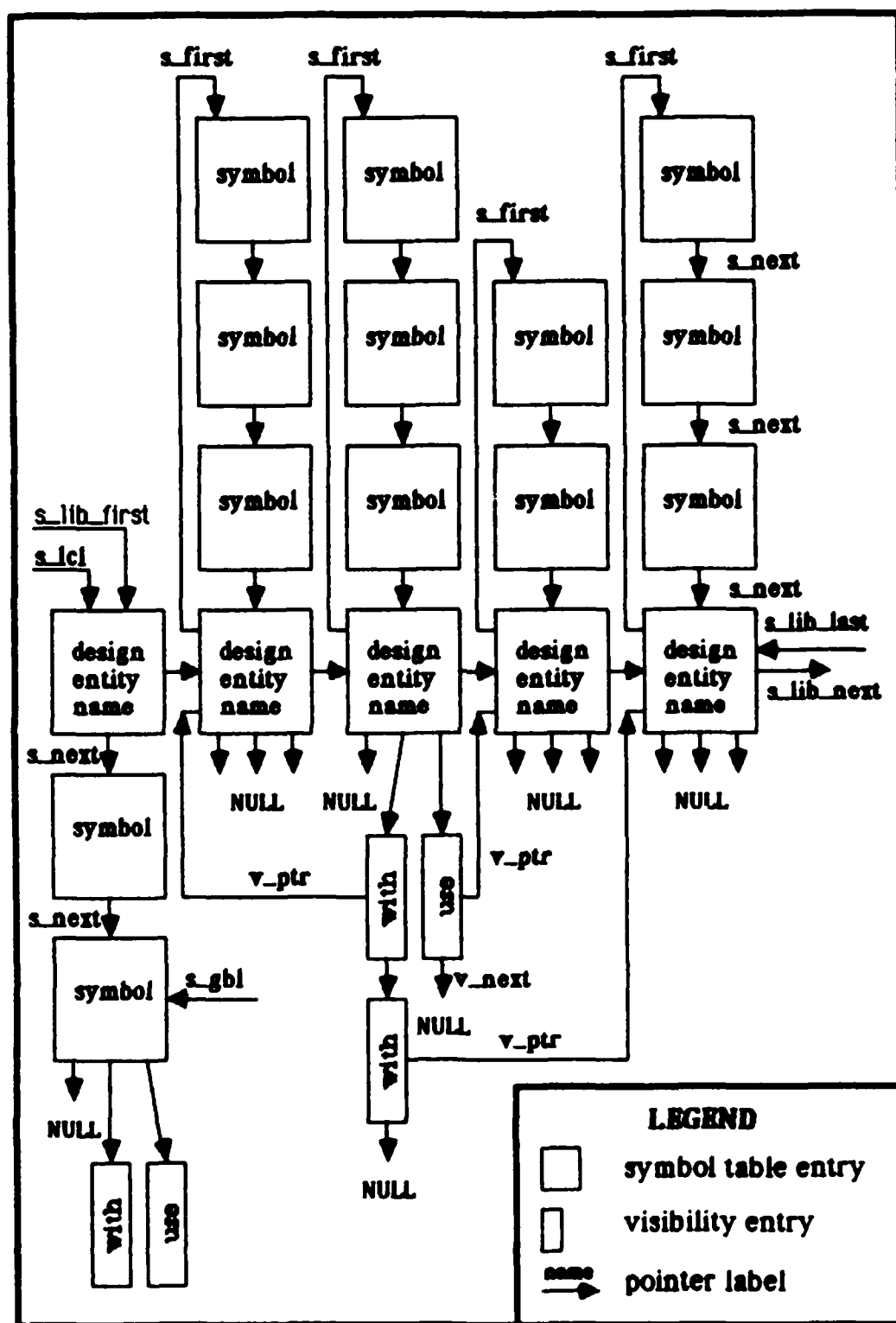


Figure 3.4: Symbol Table.

Schreiner's and Friedman's symbol table was a simple linked list (stack) of symbols, where the top of the stack contained local symbols (such as variables) and the bottom of the stack contained global symbols. The concepts of their symbol table were preserved; they are depicted in Figure 3.4 as the left-most linked list extending down the figure. In VHDL, design entities are made visible by the context clauses. Therefore, it was necessary to modify Schreiner's and Friedman's symbol table to maintain information about symbols which are not visible. The modification was accomplished by moving design entities to a second linked list for design units. The design entity linked list extends horizontally across Figure 3.4. When a context clause is specified, the appropriate design units are linked into the global region of the symbol table.

To further complicate the issue, VHDL's architectural bodies and configuration bodies can inherit direct visibility from an interface definition. This, therefore, necessitated the preservation of the information in the context clauses by the linked lists labeled *with* and *use* in Figure 3.4.

Code Generator. Since Schreiner's and Friedman's code generator targeted the C language, it was necessary to modify their code generator to create the VHDL Intermediate Access (VIA) format. The modifications made to the code generator are explained in Chapter 4.

Incremental System Implementation.

Recall, from Chapter 1, to implement the system described above an incremental development approach was chosen because three parallel

development projects required early access to the intermediate information generated by the language analyzer. The increments could have been chosen either vertically (to completely design, code, and test the language analyzer as three distinct tasks) or horizontally (to specify language subsets, each of which would be designed, coded, and tested before beginning work on the next subset). The horizontal VHDL subsets were chosen 1) to enable early identification and resolution of potential problems; 2) to ensure correct external interfaces for the AVE tools; 3) to establish correct internal interfaces for the analyzer's modules; and 4) to reduce the complexity of the project by limiting the scope of the problem.

With this decision, the question of how to subset the language arose. As defined by the Language Reference Manual (Intermetrics, 1985a: 1-4, 10-1), VHDL has five major constructs or design entities: *interfaces*, *packages*, *subprograms*, *architecture bodies*, and *configuration bodies*. Interfaces primarily define the signals, ports and other resources which define the external view of a hardware component. Packages primarily define the existence of software types, procedures and functions which are used within other subprograms and architecture bodies. Subprograms primarily define the behavior of a hardware component. Architecture bodies primarily define the structure of a hardware component. Within an architecture body, subprograms are used to describe behavior when the level of a design requires a functional description. Configuration bodies primarily define the interconnection of ports between two distinct hardware components.

To reduce the complexity of implementing the language analyzer, analyzer development was undertaken in subsets corresponding to VHDL's

design entities, context clauses, declarations, expressions, sequential statements, concurrent statements, configurations, subprograms, and other constructs. As subset were added to the language analyzer, each of the five major language constructs received enhanced capabilities.

Design Entities. The design entities subset addressed those features of the language required to recognize the semantic shells for interfaces, packages, subprograms, architecture bodies and configuration bodies (Intermetrics, 1985a: 1-1, 1-4 to 1-5, 2-1, 3-1). This subset did not address any optional semantic capabilities allowed in VHDL because they were addressed by the next five subsets.

Context Clauses. The context clauses subset addressed those features of the language required to establish the VHDL inter-entity scoping rules (Intermetrics, 1985a: 10-2). This subset did not address multiple input files because multiple input files were outside the scope of this project; but it did address multiple design entities within a single file.

Declarations. The declarations subset addresses all formal declarations allowed within any VHDL design entity (Intermetrics, 1985a: 5-1). This subset did not address the use of complex expressions to establish a declaration. Only one expression was used, a simple name; the use of complex expressions in declarations was addressed in the next subset. This subset was selected due to the commonality of declarations across all design entities, and because VHDL is a strongly typed language.

Expressions. The expressions subset addressed those features of the language required to capture and process the semantics of an expression (Intermetrics, 1985a: 7-6 to 7-20). This subset did not establish the semantic validity of concurrent or sequential statements (these concepts were addressed in the next two subsets). The subset was chosen to complete the declarations started in the previous subset.

Sequential Statements. The sequential statements subset addressed those features of the language needed to describe the function or the behavior of a hardware component (Intermetrics, 1985a: 8-1 to 8-12). This subset did not establish the validity of concurrent statements (see next subset). The subset was chosen based on its similarities to general-purpose programming languages with which the author was familiar.

Concurrent Statements. The concurrent statements subset addressed those features of the language needed to describe the structure of a hardware component (Intermetrics, 1985a: 8-12 to 8-26). This subset was chosen to complete the architectural body capabilities.

Configurations. The configurations subset addressed those features of the language needed to link architecture bodies with configuration bodies (Intermetrics, 1985a: 1-5). This subset was chosen to complete the configuration body capabilities.

Subprograms. The subprograms subset addressed those features of the language needed to link subprograms to other subprograms, packages, and architecture bodies (Intermetrics, 1985a: 2-1 to 2-4). The subset was chosen

to complete the subprogram capabilities.

Other. The other subset addressed features of the language which were potentially omitted in the previous subsets. This subset was chosen to complete the entire language capabilities. If a capability was known, but intentionally not addressed, then the deviation from VHDL Version 7.2 was explained in Appendix A.

Table 3.1 presents the capabilities projected for each of the five major language constructs as each subset was added to the language. The initial subsets provided a firm foundation across the entire language. The logical progression of capabilities across the language reduced the risk factors involved with an incremental development. These risk factors were 1) inaccurate or incomplete understanding of VHDL semantics; 2) reduced probability of a good design solution due to narrow problem focus; 3) increased probability of code modifications due to design changes; and 4) increased probability of repeating completed test analysis due to design and code modifications. The selection of the subsets in a pyramid fashion restricted these potential risks by narrowing the problem domain to either the current or the previous subset.

Intermediate Form.

As discussed in Chapter 1, three potential intermediate forms could have been used: an original design, Intermediate VHDL Access Notation (IVAN), or Design Data Structure (DDS). An original design would have extended the initial design time by at least six weeks. The extra design time was

construct subset	interface	package	subprogram	architecture body	configuration body
design-file	X	X	X	X	X
context-clause	X	X	X	X	X
declarations	X	X	X	X	X
expressions	X	X	X	X	X
sequential stmts			X	X	X
concurrent stmts				X	X
configurations				X	X
subprograms		X	X	X	
other					

Table 3.1: VHDL Subsets and Capabilities.

prohibited by the parallel AVE development efforts requiring the definition of the intermediate form for their design cycle. Although both IVAN and DDS were documented, IVAN assumed the existence of a design library manager. Since the prototype AVE did not include or require a design library manager, DDS was selected over IVAN.

An extension of DDS is the underlying structure for the intermediate form chosen. Both VHDL and DDS have constructs to hierarchically represent behavior (or dataflow), time, and structure. VHDL and DDS basically represent the same information with respect to either behavior or structure. Yet, they represent different timing information. To explain this difference, we shall say that VHDL represents *dynamic sequencing* and *dynamic scheduling* time, while DDS represents *static sequencing* time. Dynamic sequencing is the determination of the next state of a simulation model based upon the current state during execution. Dynamic scheduling is essentially the process of

specifying which of the current states will affect future states of the simulation model. With static sequencing all possible next states are determined prior to execution of the model.

Due to these differences, DDS was extended to include the dynamic sequencing and dynamic scheduling times. Additionally, DDS was explicitly designed as an interface for LISP programs. As mentioned earlier, one requirement for the AVE analyzer was to interface with a wide range of design tools. To achieve this requirement, the extension to DDS was designed to be language independent. This extension to DDS is called *VHDL Intermediate Access (VIA)* format.

The VIA format is an alphanumeric pile file (i.e., a file with sequential variable-length records) format which was developed with five underlying constraints derived from the project requirements: an incremental development approach, consistency of representation, simplicity of representation, ease of modification, and language independence. For an interface to be specified in increments, consistency of representation, simplicity of representation, ease of modification, and language independence become critical to the success of the project.

The full VIA format specification is provided in Appendix B; yet, a brief description is given here. In typical usage, the analyzer creates a VIA file in the user's present working directory. The file consists of one control record, *viatable*, followed by one or more VIA records. The format for all records is the same:

record-number record-type-name (field-name-1 = field-value-1 ; ... ;
field-name-n = field-value-n ;)

Each record starts on a new line with a positive integer record-number. The record-number of the control record is always zero. The record-number is followed by a space, then the record-type-name. The record-type-name indicates the type of the record and establishes the valid field-names which are used for that record type. Following the record-type-name is a list of field-names and field-values separated by semicolons and enclosed within parentheses. Only those field-names which have an established value (which is not a default value) are printed.

Following the field-name is the equal symbol, which indicates the field-values will follow. Each field-name has one field-value followed by a semicolon. A semicolon followed by a closing parenthesis indicates the record is complete. Any white space within a record is a delimiter, unless the white space is enclosed in quotation marks. White space includes blanks, tabs, new lines, etc.

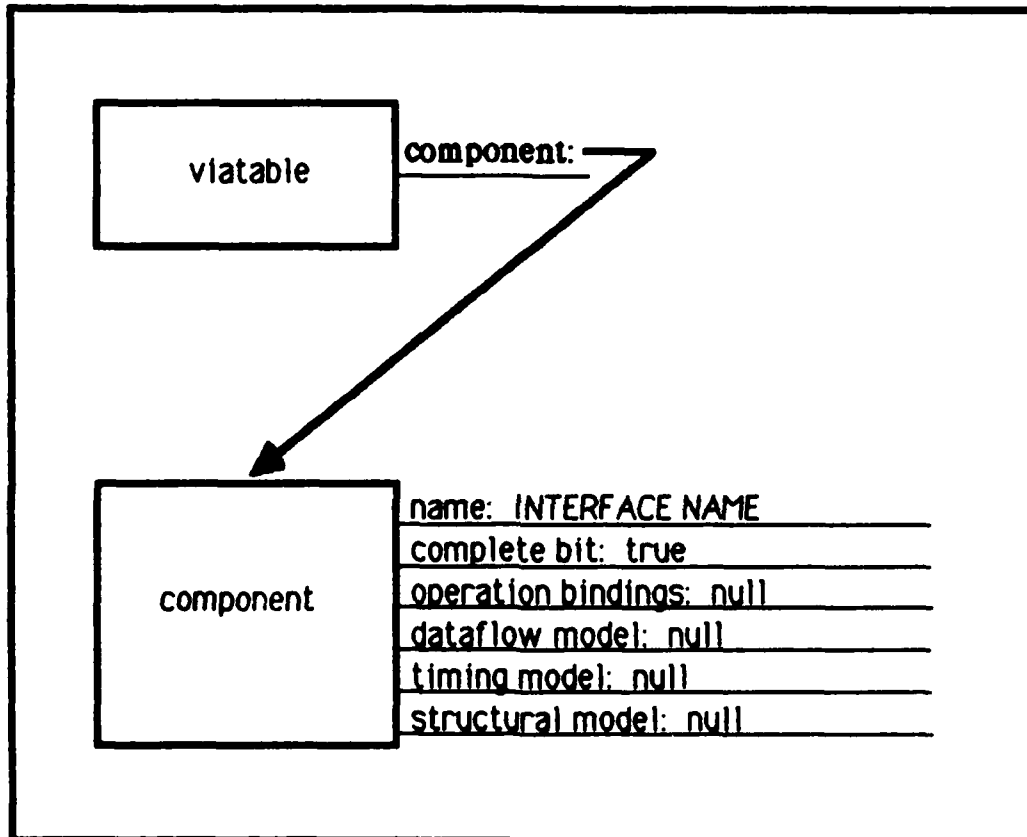
Appendix B contains a complete list of record-type-names with their associated field-names and field-value definitions; and Appendix C provides several examples, one of which is depicted in Figure 3.5.

The interface declaration depicted in Figure 3.5 was selected as an example based upon its completeness and simplicity. The first block in the figure represents a record with the record-type-name *viatable*. The *viatable* points to another record with the record-type-name *component*. This information is represented in the first VIA record just below the diagram.

VHDL Source Code:

```
entity INTERFACE_NAME is  
end ;
```

Enhanced DDS:



VIA Representation:

```
0 viatable ( component = 1 ; )  
1 component ( name = INTERFACE_NAME ; )
```

Figure 3.5: VHDL Represented in VIA.

Since the field-name is also a valid record-type-name, the field-value is interpreted as a record number, in this case the number 1. Therefore, record 1 is expected to be a component record.

The second block of the diagram points to no other blocks, but has six attributes. Of these six attributes, only *name* contains information which is not default information. Therefore, the only field-name which appears in record 1 is name. The other attributes which contain default information are not printed in the VIA file. The default values are assumed for the non-existing field-name in the VIA file.

This example, although simple, shows how the basic VIA format supports the requirement to reduce output file size. The long record-type-name and long field-names, and the redundancy between the record-type-names and the field-names render the file readable. Assuming the existence of default values minimizes the size of the overall file.

Summary.

In this chapter, nine system requirements were established to emphasize function as a primary goal and performance as a secondary goal for the language analyzer design. These nine requirements were incorporated into a system organization which increased transfer of technology and the use of computer-aided design tools. The basic design of the system was tailored after a C compiler designed by Schreiner and Friedman. Their design coupled with the use of computer-aided design tools facilitated the ease of maintenance required for an the incremental development approach. The

incremental development approach was further supported by the nine VHDL language subsets chosen to provide a wide range of capabilities across the entire language. The careful selection of subsets reduced four risk factors by insuring the scope of potential problems or of design changes was limited to either the current or the previous subset. Finally, the selection of VIA as an extension of DDS reduced the design time required to specify an intermediate data structure.

IV. Detailed Design

Overview.

The analyzer's system organization presented in Chapter 3 was designed to allow an incremental implementation based upon nine VHDL language subsets. The nine subsets were chosen to focus the problem domain into manageable slices for which the solution domain could easily be determined. Recall, from Chapter 1, after the subsets and intermediate form were selected several subtasks were established for each subset. These subtasks were:

- 1) Create detailed examples showing the relationships between the VHDL subset and the intermediate form;
- 2) Determine the appropriate design changes based upon those examples;
- 3) Implement the design changes;
- 4) Test the code using the examples produced in step 1, and
- 5) Analyze the results to determine whether the solution domain in fact satisfied not only the problem domain for the particular subset under consideration, but also previous subsets completed.

The design decisions for steps 1, 2, and 3 are presented in this chapter, while the analysis decisions for steps 4 and 5 are presented in Chapter 5. Now, discussing nine subsets with at least three topics each would overendow the reader with unwarranted details. Therefore, this chapter presents the important information in four sections: the basic methodology used for each subset, an example of the design work, the major design

decisions, and the language analyzer's detailed design.

Basic Methodology.

The detailed design approach applied to each subset was to create detailed examples, to determine appropriate design changes, and to implement the design changes. Each of these steps is discussed below.

Create detailed examples. Creating detailed examples of the VHDL-to-VIA relationship was the process of identifying that portion of VHDL's grammar under consideration for a particular subset, writing examples of VHDL source code, representing the VHDL code in the enhanced Design Data Structure (DDS) (Afsarmanesh and others, 1985), and translating DDS to VIA. At the beginning of this project each of the above steps was performed to assure consistency in representation and to create a firm foundation on which to build later subsets. Further into the project, translating the entire VHDL example into DDS became cumbersome and hard to read; therefore, only that portion of VHDL under consideration was translated into DDS. Appendix C contains examples which were the result of this step.

Determine the appropriate design changes. Based upon the examples generated in the previous step, detailed design changes were derived. Determining these design changes was a logical outgrowth of the first step, in that the original examples implied a pair of specific functional transformations, g and f . $f(x)$ was defined as the transformation from VHDL to DDS where x was example VHDL source code, $g(z)$ was defined as the

transformation from DDS to VIA where z was example DDS representation, and $y = g(f(x))$ was the example VIA.

The identification of these two separate transformations had two significant consequences. First, it implied that the VIA file should be created using two steps, rather than one. And second, this characterization implied that a design translation could, in fact, be performed consistently independent of the particular subset under consideration. As stated in Chapter 3, the overall design of the language analyzer was based upon Schreiner's and Friedman's work (Schreiner and Friedman, 1985). Therefore, in order to capitalize on technology transfer, the first transformation, f , was defined as an extension of the basic concepts presented by Schreiner and Friedman with respect to transforming semantic content of the source code into a symbol table. Yet, if their concepts were preserved, then either a transformation from the symbol table structure to a DDS structure was required for VIA generation (or, alternatively, the VIA generator itself could make that transformation). Traditionally, code generators do not modify input, because they are procedures which print output based upon the input. Yet, creating a third function to transform the symbol table structure into the DDS directed acyclic graph would increase memory requirements and reduce operational performance. Additionally, the above mathematical analysis suggested only two transformations were required, not three. Therefore, for the above reasons, the second technique was used, a technique which allowed the code generator to transform input, print results, and return an indicator of the result. Once established, this technique enabled the design of the language analyzer to proceed smoothly from subset to subset.

Implement the design changes. After each design increment was determined, pseudocode was written, then the pseudocode was transformed into the language C. Aside from being a good programming technique which leads to structured code modules, the author chose to use pseudocode for two reasons. First, at the beginning of this project (prior to developing the initial pre-prototype parser) the author had never written C programs. Second, it has been suggested that to learn a new language one must use what he/she knows about other languages and transform this information into the syntax of the target language (Drew, 1981). Therefore, it seemed reasonable to start with pseudocode and transform the pseudocode into C code as a secondary process. The pseudocode applied to this project was an informal technique, and, as such, is not presented in this report.

Design Work Example.

As a detailed example of the type of design work associated with the basic methodology, consider the design of the first subset, the *Design-File Problem*. The Design-File Problem was the problem of recognizing the semantic shells for interfaces, packages, subprograms, architectural bodies and configuration bodies within the *design-entity* subset. Recall, from Chapter 3, each VHDL subset was selected to provide a wide range of capabilities across the breadth of the language. Since VHDL has five principal constructs, recognizing each of these constructs seemed the ideal first problem.

Two options were available for the resolution of this problem: the five major constructs could have been designed either as individual constructs, or as a group of constructs. Processing each construct separately provided the advantage of an early understanding of how code generated by either YACC interfaced with modules designed by the author, and the disadvantage of potentially inconsistent VIA representations for the constructs. Processing the five constructs in parallel provided the advantage of consistent VIA representation and utilization of common functional modules, and the disadvantage of a potentially incomplete understanding of the YACC interface. Since neither method seemed to be ideal for all decomposition steps, a mixture of the two was decided upon for this subset. The constructs were considered in parallel for creating the detailed examples, and they were considered individually for the design and implementation steps. This decision assured a consistent VIA representation, and through it the author quickly learned techniques to interface with YACC.

As mentioned earlier, each of the subproblems needed to address three steps in the design process: created detailed examples, determine the appropriate design changes, and implement the design changes. The decisions associated with these step are discussed next.

Create detailed examples. Creating detailed examples involved four steps: identifying that portion of VHDL's grammar under consideration, writing examples of VHDL source code, representing the VHDL examples in the enhanced DDS, and translating DDS to VIA. The portion of VHDL's grammar which applied to the first subset (as depicted in Figure 4.1) was not a

1. entity INTERFACE_NAME is
 ...
end;
2. configuration CONFIGURATION_NAME
 of ENTITY_NAME for ARCHITECTURE_NAME is
 ...
end;
3. package PACKAGE_NAME is
 ...
end;
4. procedure PROCEDURE_NAME is
 ...
begin
 ...
end;
5. function FUNCTION_NAME return A_TYPE_NAME is
 ...
begin
 ...
end;
6. architecture ARCHITECTURE_NAME of ENTITY_NAME is
 ...
end;

Figure 4.1: Subset 1 -- the Design Entity Shells.

complete set with respect to VHDL's syntax. By comparing the contents of the figure to VHDL's grammar (Intermetrics, 1985a: 1-1, 1-4 to 1-5, 2-1, 3-1) one can see that the interface, listed in item 1, and the configuration body, listed

in item 2, are complete, but the other constructs are incomplete. Recall, from Chapter 1, that a pre-prototype parser created prior to the start of this project recognized all of VHDL's syntax. Therefore, although the semantic analysis was limited to the subset under consideration, correct and syntactically complete VHDL examples were created for use with this pre-prototype parser. This decision proved useful, in that, the parser's design was altered only by semantics from one subproblem to the next.

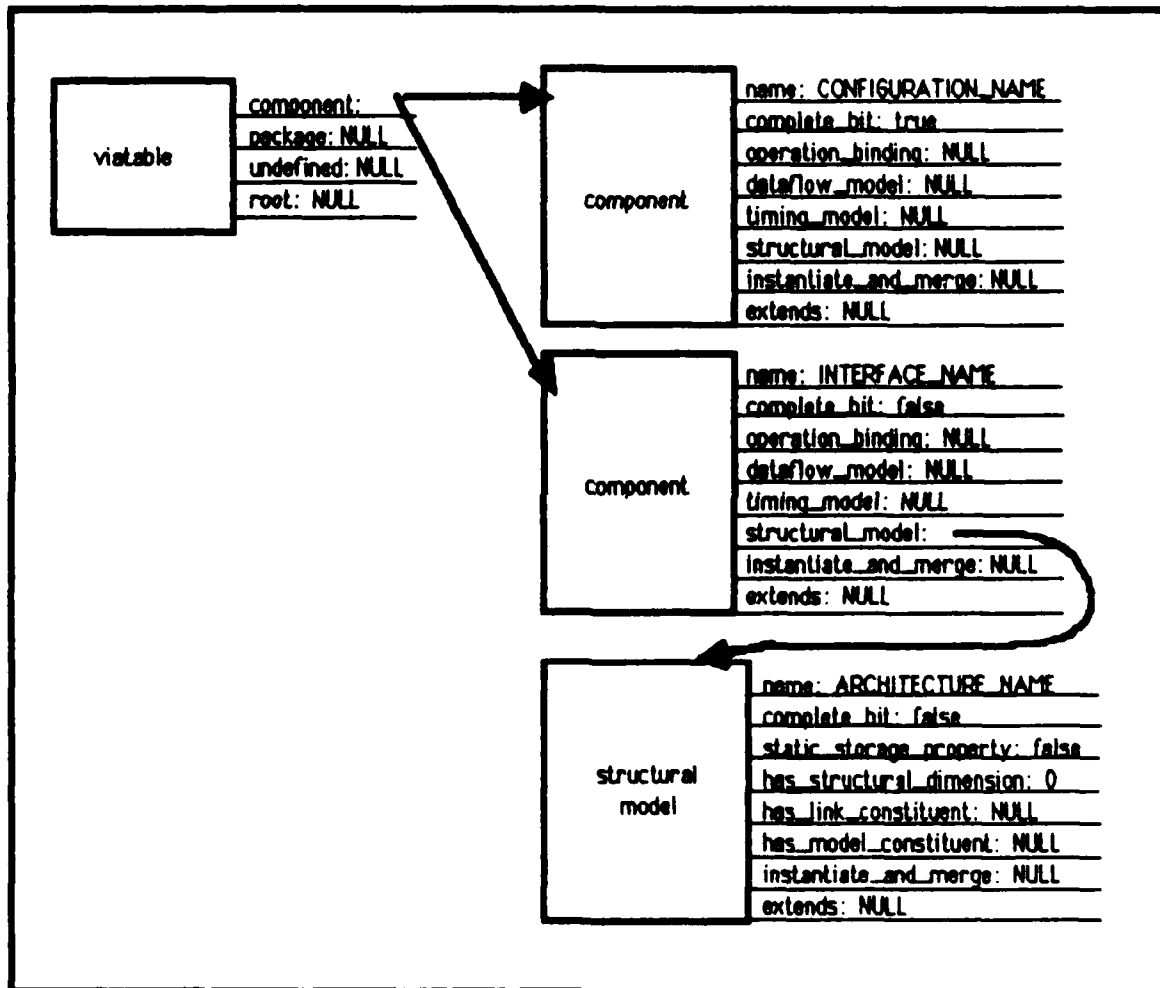
The set of examples which were generated for the Design-File Problem are listed in Appendix C. Those examples visually reflect the decisions on how to represent VHDL in the enhanced DDS and in turn in VIA. One example will be discussed to explain the interpretation of those examples. The configuration body example as reflected in Figure 4.2 was selected for discussion for two reasons. First, the configuration body was one of the two VHDL constructs for the design-entity subset, which was both syntactically and semantically complete. Since the interface example was discussed in Chapter 3, to avoid redundancy the configuration body was the logical choice. Second, the configuration body was a simple, but non-trivial, example.

Figure 4.2 embodies three sources of information: the VHDL source code, the enhanced DDS, and the VIA representation. Mathematically, the example represents a dual transformation, $g(f(\text{VHDL})) = \text{VIA}$. The VHDL source code represents the input domain; the enhanced DDS represents the application of a single transformation; and the VIA notation represents the output range. The VHDL Language Reference Manual (Intermetrics, 1985a) explains the VHDL source code, and the VIA notation was explained in Chapter 3. Accordingly,

VHDL Source Code:

```
configuration CONFIGURATION_NAME  
  of INTERFACE_NAME  
  for ARCHITECTURE_NAME is  
end ;
```

Enhanced DDS:



VIA Representation:

```
0 viatable ( model = 1 ; model = 2 ; )  
1 model ( name = CONFIGURATION_NAME ; )  
2 model ( name = INTERFACE_NAME ; structural_model = 3 ; complete_bit =  
false ; )  
3 structural_model ( name = ARCHITECTURE_NAME ; complete_bit = false ; )
```

Figure 4.2: The VHDL Configuration in VIA.

the following paragraph presents only a discussion of the transformation.

In the example, the VHDL source code used three identifiers, each of which is represented in DDS. As stated in Chapter 2, DDS is principally composed of models (or components) which have four subspaces: dataflow, timing, structural, and physical. Therefore, each of the three VHDL identifiers was mapped into either a component or one of its subspaces. Also recall, from Chapter 2, a configuration is the design entity which describes a chip's port connections for the ports specified in an interface declaration, while the architectural body describes the structure and behavior of the chip. In a complete VHDL source description of an electronic circuit, the identifier `INTERFACE_NAME` would have an associated interface declaration, and the identifier `ARCHITECTURE_NAME` would have an associated architectural body. Since `ARCHITECTURE_NAME` implies that somewhere there exists (or will exist) an architectural body which describes the characteristics of the structure of the component, the identifier was mapped into the structure subspace of VIA. Also, `INTERFACE_NAME` implies that somewhere there exists (or will exist) an interface declaration which describes the ports of the model; therefore, the identifier was mapped into DDS as a component model. Furthermore, the concept of configuration implies that the electronic circuit has more than one sub-part, otherwise there would be nothing to configure; therefore, `CONFIGURATION_NAME` was also mapped into DDS as a component. The tree in Figure 4.2 depicts these relationships.

Determine the appropriate design changes. At the beginning of the Design-File Problem, the approach explained in the earlier section, labeled

"Basic Methodology", had not been completely formalized. At the time, the principal concern for the initial design changes was centered around formulating a better understanding of YACC's interface, rather than identifying a well-founded design approach. The basic design approach used was to transform the VHDL source file into the symbol table and from the symbol table create the VIA file as the semantic content of the VHDL source was derived. This was not a good approach, although traditional. The approach led to code generation routines which were based upon the structure of VHDL for generating the directed acyclic graphs represented in a VIA file, rather than upon a more general class of code generation modules. Although the author was aware the modules would eventually require changes, the traditional approach was pursued to learn about the specific shortcomings and to avoid incorporating them into a refined approach.

Implement the design changes. The interface to YACC is not easily readable. YACC requires partial C code segments to be inserted within the YACC source file which describes the VHDL parser. YACC in turn generates a C program for the parser using the C code segments. To increase the readability of the C source file, procedure and function calls were used for the C code segments (see Figure 4.3) rather than performing actions in line. Although self-explanatory names were used, the actual names of the parameters passed by YACC were less readable than desired. As Figure 4.3 shows, YACC parameter names take the form of a dollar sign followed by a number, such as \$1 or \$2. The name \$1 means the result associated with the first non-terminal (or terminal) of the current production; the name \$2 implies the second non-terminal; and so forth. Also as shown in Figure 4.3, the C code

```

architectural_body_declaration
: ARCHITECTURE
  Identifier
  {
    make_arch( $2 );
    blk_push();
  }
  OF
  library_name_or_id
  {
    process_visibility( $5, INTER );
  }
  IS
  block_statement
  ENDRW
  option_simple_name
  {
    verify_names( $2, $10 );
  }
  Semicolon
  {
    architectural_body_declaration( $2, $5, $8 );
    delete_visibility( $5 );
    change_regions( $2 );
  }
  ;

```

Figure 4.3: An Example Parser Production for YACC

segments are inserted between two non-terminals, or between a non-terminal and a terminal. Each time a set of C code segments is inserted into a production, YACC generates a new empty non-terminal for that segment. This implies when new C code segments are inserted the subsequent numbers change, so the parameter numbers must also change. Therefore, to reduce potential errors, with respect to proper numbering, modification of productions should be kept to a minimum.

In summary, completing the Design-File Problem not only provided the analyzer with its first semantic capabilities but also influenced future design decisions by:

1. establishing basic relationships between VHDL and VIA.
2. realizing that for any particular subproblem the principal focus is on a specified subset of VHDL. These subsets must be viewed in the context of the closely associated VHDL statements (such as the architectural bodies association to the block statement.)
3. identifying the need for consistency in the design of code generation routines.
4. acquiring a firm understanding of the YACC interface.

Major Design Decisions.

The basic approach used for the Design-File Problem was applied to the other subset problems which were completed. The incremental approach to designing the language analyzer surfaced many design decisions. Since defining each of these design decisions would once again overendow the reader with details, a representative set of major design decisions is presented. These decisions include:

1. Extend the design of the symbol table.
2. Create VIA generation routines based upon attributes.
3. Establish parser interfaces for all subsets.
4. Create and use abstract data types.

Each of these decisions is described in the following paragraphs.

Extend the design of the symbol table. Although Schreiner's and Friedman's symbol table was useful for the Design-File Problem, their simple symbol table did not support the various VHDL symbol types or scope requirements. As discussed in Chapter 3, Schreiner's and Friedman's symbol table was extended to support direct, indirect, and inherited visibility. The symbol table was also extended to support VHDL type and subtype definitions, all variables, signals, and literals (i.e., decimal-literals, based-literals, abstract-literals, character-strings, and bit-strings). As new symbol characteristics were discovered, new fields were added to the symbol table. Since the language analyzer is a prototype, the symbol tables' data structure was not optimized because, as stated in Chapter 1, this thesis project emphasized function, with performance as a secondary goal.

Create VIA generation routines based upon attributes. In solving the Design-File Problem, the symbol table was used to store directed acyclic graph (DAG) information which is represented by the VIA records. As the research progressed, the author realized that although the symbol table stores VIA information which directly relates to a symbol (i.e., name, descriptive characteristics and record location), the vertices in the symbol tables' linked lists do not correspond one-to-one to the vertices in the VIA DAG. Many VIA records are printed without a name, yet only those records with names could be represented in the symbol table. Additionally, most attributes in a VIA record are pointers to other VIA records. Therefore, the data type *attribute*, a linked list representing partial DAGs, was created and used by the VIA generation routines.

Establish parser interfaces for all subsets. The implementation transition from the Context-Clause Problem to the Declarations Problem required interfaces for the new parser actions. When these interfaces were added to the parser, YACC did not produce an error-free parser. Upon investigation, some declaration grammar rules were used by expressions. Since expressions were targeted for a later subset, the impact of the interfaces on the expressions subset was not considered during the design of the Declarations Problem. To resolve this interface problem (and future problems of a similar nature) the YACC VHDL source description file was changed to include *stubbed interfaces* for every production in the source description. Stubbed interfaces are function calls to functions which have names identical to the production names. These functions, when initially written, included one parameter for each production data item available, printed a "production not implemented" statement, and returned a null value. These stubbed interfaces not only allowed YACC to produce an error-free parser, but also reduced future changes to YACC's VHDL description, and thereby reduced overall development time.

Create and use abstract data types. The term *abstract data type* is defined (e.g., Fairley, 1985: 96) as a data structure and its associated operators (i.e., functions and procedures). By creating and using abstract data types with their associated operators, the implementation details of the data structures are separated from the implementation details of the program flow and thereby facilitate information hiding principles. Three abstract data types were defined: symbol table, attribute table, and group. Each of these abstract data types is defined in the next section.

Language Analyzer Detailed Design.

The dataflow diagram for the language analyzer which evolved during the design process is depicted in Figure 4.4. This dataflow diagram presents six high-level data transformations: get tokens, find next state, process actions, process symbol table, process attribute table, and build groups.

Get Tokens. Get tokens is the process of lexical analysis by which LEX (Lesk and Schmidt, 1978) identifies *tokens* (such as keywords, identifiers, literals, punctuation marks and operator symbols) from a VHDL source code file. These tokens are in turn used for two other processes (find next state and process symbols) which will be described later. The basic *get token* process consists of scanning the input file (one character at a time) to match the longest pattern which describes a token. The pattern for a keyword token is a proper subpattern of the pattern for an identifier token; therefore, these are selectively differentiated by a binary table lookup process. The token is compared to the entries in a reserved word table containing a complete list of VHDL keywords. When the token is found in the table, the token is considered a keyword; otherwise, it is assumed to be an identifier. All tokens are passed to the *find next state* task, while only identifiers and literals are passed to the *process symbols* task.

Find Next State. *Find next state* is the parsing process by which YACC (Johnson, 1978) determines the next production state based upon the current production state and the token received from the *get token* task. This

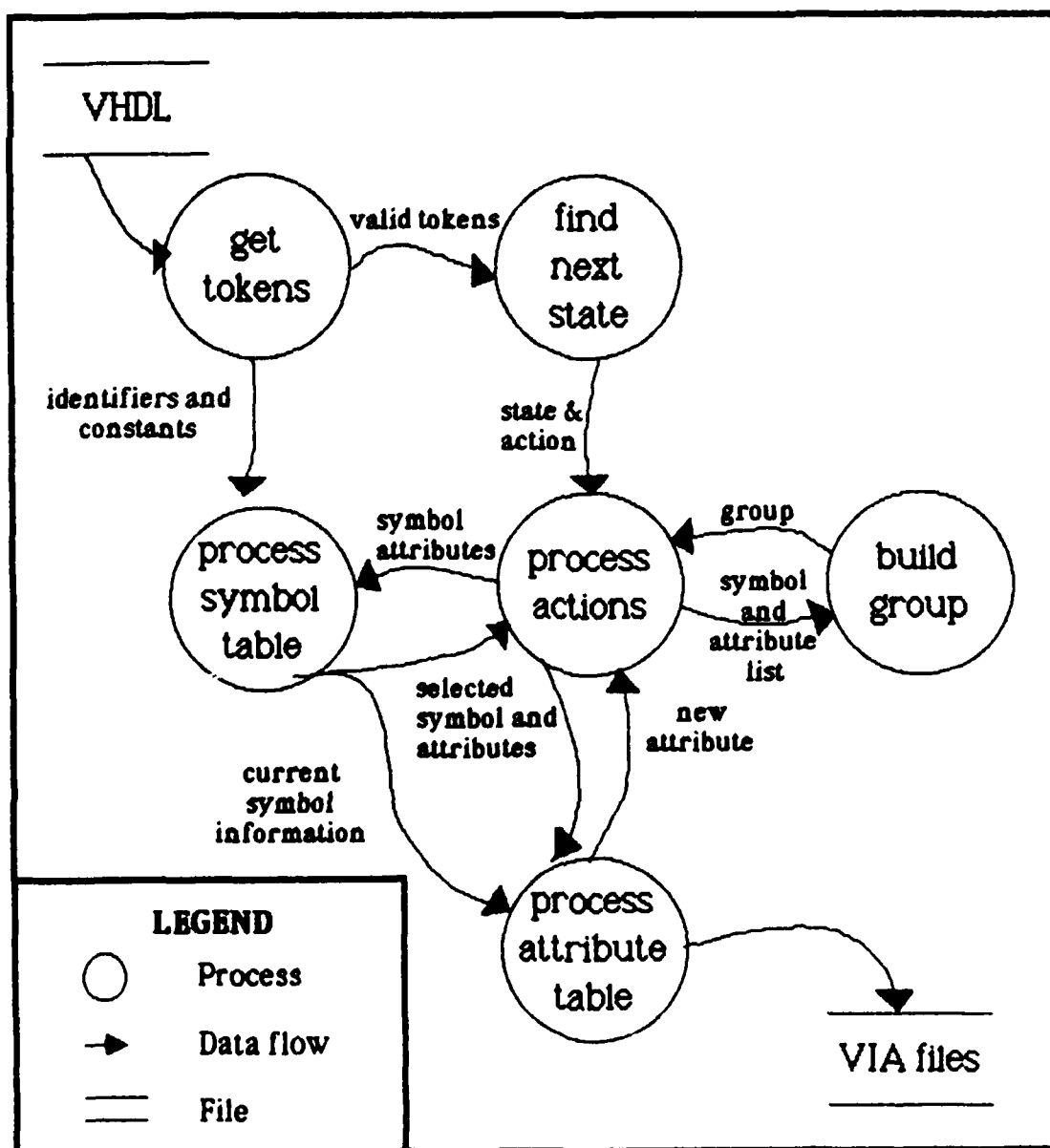


Figure 4.4: High-level Dataflow Diagram of the Language Analyzer.

process is automatically generated by YACC based upon a set of VHDL productions which are similar to the grammar rules listed in the VHDL Language Reference Manual (Intermetrics, 1985a). Associated with each production is a production name and a set of actions which are executed upon

recognition of the production. Upon recognizing a particular production, the current state information (e.g., identifiers used in the production, and keywords like STATIC, ATOMIC and so forth) is passed to the *process actions* task for appropriate resolution of the action.

Process Actions. *Process actions* is a conglomerate task with multiple subtasks. Each subtask has a unique name which corresponds to one production name in the *find next state* process. Each subtask performs a set of actions based upon the production specifications in the VHDL Language Reference Manual (Intermetrics, 1985a). These actions are primarily control actions which monitor the symbol table processing and attribute table processing, yet occasionally these actions group together symbol table and attribute table information. The symbol table is an abstract data type used to maintain characteristics of identifiers and literals, while the attribute table is an abstract data type used to maintain characteristics of a directed acyclic graph (DAG) represented by the VIA records. Both abstract data types are discussed in the next two sections.

Process Symbol Table. *Process symbol table* is a set of tasks associated with maintaining the abstract data type symbol table. The symbol table consists of two different objects (symbols and vislinks) and three classes of operations (constructors, mutators, and observers). Each VHDL identifier and literal (i.e., constants, strings and so forth) is assigned to a unique symbol. Additionally, any identifier used in a VHDL *context clause* is assigned to a unique vislink, which defines the scope of a symbol's external visibility. These symbols and vislinks are linked together as described in

Chapter 3. Each symbol contains characteristics which describe how the symbol is used in VHDL, how the symbol is linked into the symbol table, and how the symbol was used in VIA. The information associated with the VHDL usage consists of characteristics such as name, type (i.e., ARCH, CONF, PORT, SIG, VAR), value (i.e., values associated with VHDL type definitions, and constants), variable type, signal type, and so forth. The information associated with the symbol table linkage consists of characteristics such as next symbol, last symbol, next design entity (i.e., any symbol derived from an architecture, configuration, package, interface, or subprogram name), last design entity, use links (i.e., symbols which were declared in a VHDL *use clause*), with links (i.e., symbols which were declared in a VHDL *with clause*) and so forth. The information associated with the VIA usage consists of characteristics such as *complete_bit* (i.e., a flag indicating a symbol was completely defined), *static* (i.e., a flag indicating a static signal), *duration* (i.e., an indicator showing how long a signal is available), and so forth.

All of these characteristics are manipulated by a set of operations consisting of *constructors*, *mutators* and *observers* (Fairley, 1985: 98). Constructors are those operations which create the symbols or vislinks, such as *s_create* which creates a symbol. Mutators are those operations which alter the contents of symbols or vislinks, such as *remove_vislink* which removes a particular vislink. Finally, observers are those operations which retrieve information from the symbol table without modification, such as *s_find* which finds a particular symbol.

Process Attribute. *Process attribute table* is a set of tasks associated with maintaining the abstract data type attribute table and with generating the attributes associated with the VIA DAG. The attribute table consists of one object, called an attribute, and the same three classes of operations associated with the symbol table. As depicted in Figure 4.5, the attribute table is a simple linked list. Each attribute contains a pointer to the next attribute, a record type identification, and a record number. The record type identification and the record number are integers which represent the *record_type_name* and the *record_number* for the VIA record which was printed when the attribute was created.

The attribute table is essentially an ordered linked list with the most recently printed VIA record at the front of the list. VIA records are printed in a hierarchical order, with the lowest level printed first. At the time VIA records are printed the attributes are entered into the attribute table. They are preserved in the attribute table until all other VIA records which reference them are printed. For example, when a *single_value* record is printed, an attribute is entered into the attribute table. This attribute is maintained in the attribute table until both a *dataflow_model* record and a *dataflow_link* record are printed and their respective attributes are entered into the attribute table. At that time, the attribute for the original *single_value* record is no longer required and is therefore removed. Additionally, when a VIA record is printed with the field_name *name*, the symbol table routine *update_where* enters the record number into the symbol table. This allow VIA records which are associated with a VHDL identifier or constant to be referenced after the attribute has been removed from the

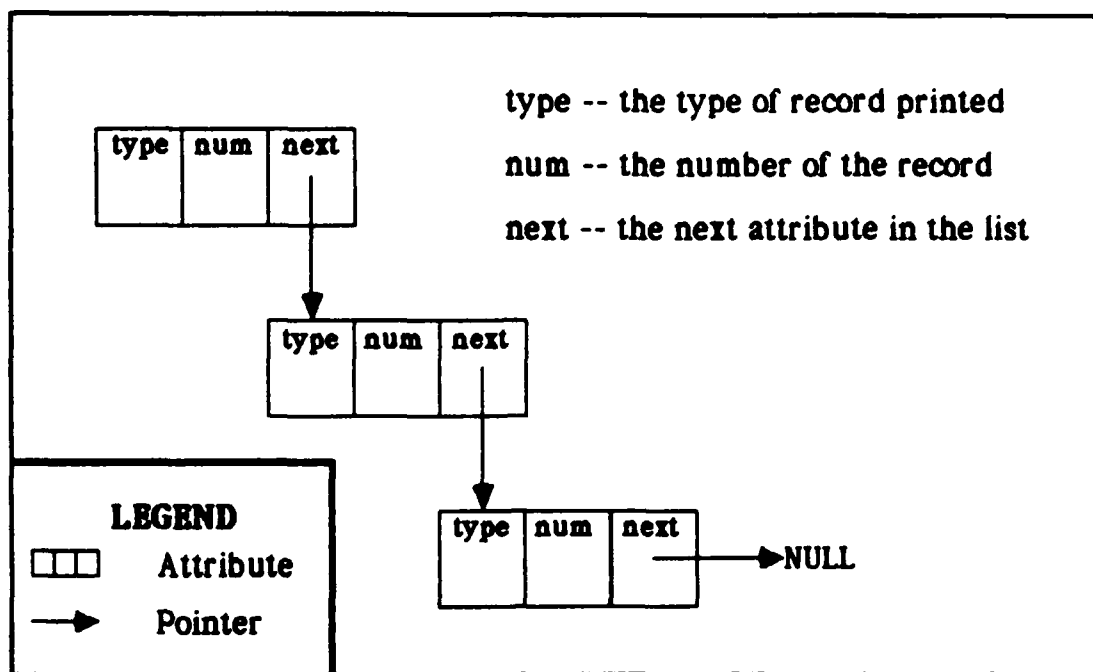


Figure 4.5: Attribute Abstract Data Type.

attribute list.

Build Groups. *Build groups* is the process of associating a specific symbol table entry and attribute table entry together. YACC allows only one data type per production. Yet the actions for particular productions require the information from both the symbol table and the attribute table. To avoid the use of global variables, and to preserve the definition of an abstract data type during implementation, a third abstract data type called a group was developed. The group data type is a simple one node data type which contains a pointer to a symbol table entry and a pointer to an attribute table entry as depicted in Figure 4.6. The group data type's sole purpose is to fulfill YACC's one-data-type-per-production requirement. Therefore, the *build group* process consists of the operations which can be performed upon the data type

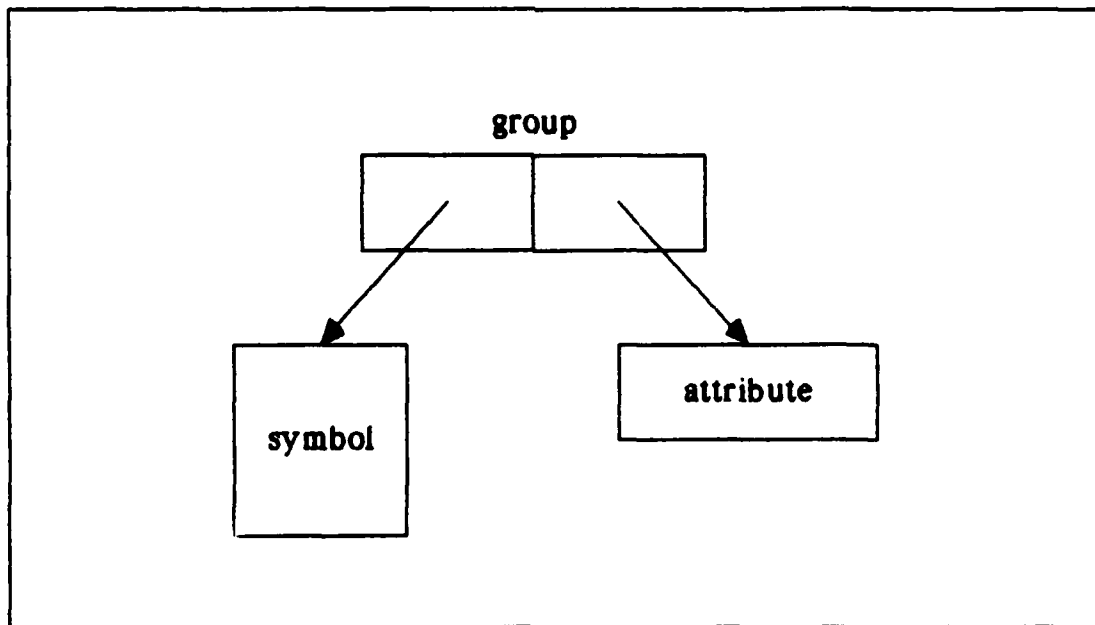


Figure 4.6: Group Abstract Data Type.

group: create a group, delete a group, get a symbol table pointer, and get an attribute table pointer. *Create_group* allocates memory for the abstract data type and updates the associated pointers. *Free_group* frees memory. *Get_symbol* retrieves the groups symbol table pointer, while *get_attribute* retrieves the attribute table pointer.

Summary.

This chapter discussed the basic methodology for three of the five remaining subtasks discussed in Chapter 1. These subtasks were to create detailed examples, to determine design changes bases upon those detailed examples, and to implement the design changes. The design work associated with the Design-File Problem was discussed to explain the types of design

decisions associated with the three subtasks. The solution to the Design-File Problem recognized the semantic shells for VHDL's architecture bodies, configuration bodies, packages, interfaces, and subprograms, and produced the VIA records describing those constructs. The completion of the Design-File Problem influenced future design decisions by establishing the degree of complexity associated with determining VHDL-to-VIA relationships and with implementing those relationships.

After discussing the Design-File Problem, this chapter described several fundamental design decisions derived from processing other VHDL subsets were discussed. Extend the design of the symbol table and create VIA generation routine were among these decisions. The design of the basic software modules for processing the symbol table, processing the attribute table, and building groups were discussed. These software modules, coupled with established interfaces for each parser action, provide a firm foundation for continued design work. The next chapter discusses testing of these software modules. The test results demonstrate that the modules work as designed, and therefore reduce future testing complexity.

V. Analysis

Overview.

The language analyzer's detailed design, as presented in Chapter 4, was selected to fulfill the eight system requirements listed in Chapter 3. These system requirements were 1) to embed the language analyzer in the UNIX environment; 2) to support a wide range of VHDL design tools; 3) to analyze the syntax and semantics of VHDL; 4) to emphasize user friendliness; 5) to facilitate ease of maintenance; 6) to process a 1000-line input file within three minutes of CPU time; 7) to analyze one input file per execution of the language analyzer; and 8) to reduce output file size. To assure these system requirements were fulfilled, individual test requirements were derived to verify the design implementation. Based upon the test requirements, specific tests were selected and performed. This chapter presents the test requirements and examples of particular tests with their associated results.

Test Requirements.

The language analyzer's design implementation was to be considered correct and complete when the language analyzer successfully satisfied the following test requirements:

1. Analyze a representative sample of arbitrary VHDL source code files.
2. Analyze a single 1000-line VHDL source code file within three CPU minutes on a VAX-class machine.
3. Create valid VIA file contents for representative sample sets.

4. Produce valid error and warning messages for representative sample sets.
5. Operate within the UNIX environment.

One other test requirement was considered, but not included: the requirement to interface with other VHDL tools in the AVE. As mentioned in Chapter 1, four VHDL tools (a VHDL code checker, a microcode compiler, a software simulator, and a simulator generator) were scheduled to be developed in parallel with the language analyzer. Of these four tools, the microcode compiler, the software simulator, and the simulator generator were *designed* in parallel; yet, their implementations were delayed enough that testing of the actual interface could not be accomplished within the schedule of this project. Therefore, interfacing was not included as a requirement for the prototype language analyzer.

Each selected test requirement is discussed below.

1. Analyze a representative sample of arbitrary VHDL source code files.

The ability to analyze an arbitrary VHDL source code file partially fulfills three system requirements: to embed the language analyzer in the UNIX environment, to emphasize user friendliness, and to analyze one input file per execution of the language analyzer. UNIX software products are generally allowed to receive input from any arbitrary file or group of files. Although for the purposes of this project the language analyzer was restricted to one input file, allowing this to be an arbitrary file was consistent with UNIX conventions, and, therefore, presumably user friendly. In particular, the user

is not required to learn new techniques for handling input files.

2. Analyze a single 1000-line VHDL source code file within three CPU minutes on a VAX-class machine. The analysis of a single 1000-line VHDL source code file within three CPU minutes was directly stated as a system requirement. As indicated in Chapter 1, this requirement was established to provide a minimal acceptable performance baseline for the prototype language analyzer.

3. Create valid VIA file contents for representative sample sets. The requirement to create valid VIA file contents was derived from two system requirements: to support a wide range of VHDL design tools; and to analyze the syntax and semantics of VHDL. The VIA file generated by the analysis of the syntax and semantics of VHDL must be correct as specified in Appendix B.

4. Produce valid error and warning messages for representative sample sets. Producing valid messages is a test requirement derived from three system requirements: to analyze the syntax and semantics of VHDL, to emphasize user friendliness, and to facilitate ease of maintenance. If input file contents are incorrect, then specific, concise error or warning messages are generated by the language analyzer.

5. Operate within the UNIX environment. This test requirement follows directly from the system requirement to embed the language analyzer in the UNIX environment. In addition to following UNIX input conventions (discussed earlier), the language analyzer fulfills this requirement by permitting

multiple users to simultaneously execute the analyzer.

Method of Evaluation.

According to Fairley (Fairley, 1985: 184-185), four classes of tests should be performed on any software product: *functional*, *performance*, *stress*, and *structural* tests. Functional tests verify post-conditions based upon a selected set of pre-conditions (i.e., including conditions "inside, on, and just beyond the functional boundaries" (Fairley, 1985: 271). Performance tests verify particular interactions of the software product with its environment. Stress tests attempt to overload a system and its environment in order to establish operational limits and failure reasons. Finally, structural tests verify internal program logic, assuring each logical path within the code functions correctly. Of these four classes of tests, formal test cases were established for functional and performance tests. Test cases for stress and structural tests were not formally specified for the reasons discussed below.

Functional tests. For the reasons discussed in Chapter 1, functional testing was given the highest priority among the various test classes. Prior to designing program modules for each subproblem (see Chapter 4), a representative set of functional test cases was developed. For each test case, VHDL source code was written and expected results were predicted for comparison with analyzer output. Two kinds of test cases were developed: single and multiple grammar rule tests. The single grammar rule test cases contained VHDL source code to demonstrate one particular aspect of the VHDL grammar subset under consideration. By limiting test input to exhibit one

grammar rule, the expected results were more easily verified. With the multiple grammar rule test cases, various combinations of the grammar rules were created to assure the results were not affected by second-order effects. By selecting test cases in this manner, the grammar rules from a previous subset could be used in the test cases for a current subset with a degree of confidence that the expected results from the previous subset would be stable. The test cases making up this representative sample set appear in Appendix C.

Performance tests. For the reasons discussed in Chapter 1, performance testing was given a low priority, although formal test cases were selected to demonstrate the prototype language analyzer performance characteristics. The original design goal projected the analyzer would execute a representative 1000-line VHDL source code in less than three CPU minutes on a VAX class machine. Since the prototype language analyzer processes only a portion of the VHDL language, test were created to determine the performance based upon the implemented subsets. These tests consisted of analyzing the VHDL source code files under normal operating conditions of the AFIT host UNIX environment. Run time data for ten test iterations of four sets of VHDL code was gathered and statistically evaluated to determine the mean execution time and standard deviation. Based upon these results, projections for the completed prototype language analyzer are presented.

Stress tests. Stress testing was given a low priority in order to minimize the level of effort associated with testing the language analyzer within a fixed length of development time. (As stated in Chapter 1, the

primary emphasis of this research endeavor was to establish a functionally correct prototype language analyzer with performance, including stress performance, as a secondary consideration.) To establish that the language analyzer would not fail in an operational environment, however, students of two AFIT Computer Architecture classes were assigned homework which required developing VHDL source descriptions which were processed by the language analyzer. This informal testing provided stress test coverage adequate for the purposes of this project.

Structural tests. Structural testing was given a low priority for two reasons. First, structural tests are designed to test a program's logic paths, thereby assuring each statement operates as intended. In most cases, the same result can be derived during functional testing, given that the inputs for the functional tests are carefully selected to achieve this goal. Second, the language analyzer is a prototype. As a prototype, the language analyzer's code is subject to change as the development progresses. Therefore, provided the language analyzer passes the functional tests (which demonstrates working code), the potential time spent on performing structural tests could be used for continued design work, although some modules were selected for structural testing based upon the complexity of the module and upon how frequently the module was used by other program modules. For instance, the attribute modules which processed VIA attributes and wrote VIA records were structurally tested because approximately 75 percent of the action modules call the attribute modules.

Once a program module passed a structural test, the module was considered correct, and that module was used to test other program modules. Such a bottom-up approach to structural testing allowed composite program modules to be tested without physically modifying them or writing additional test routines.

Evaluation of the VHDL Language Analyzer.

The evaluation of the language analyzer consisted of the four classes of tests previously mentioned. The description of the test, the test conditions, and an evaluation of the test results are presented below for the tests in each class.

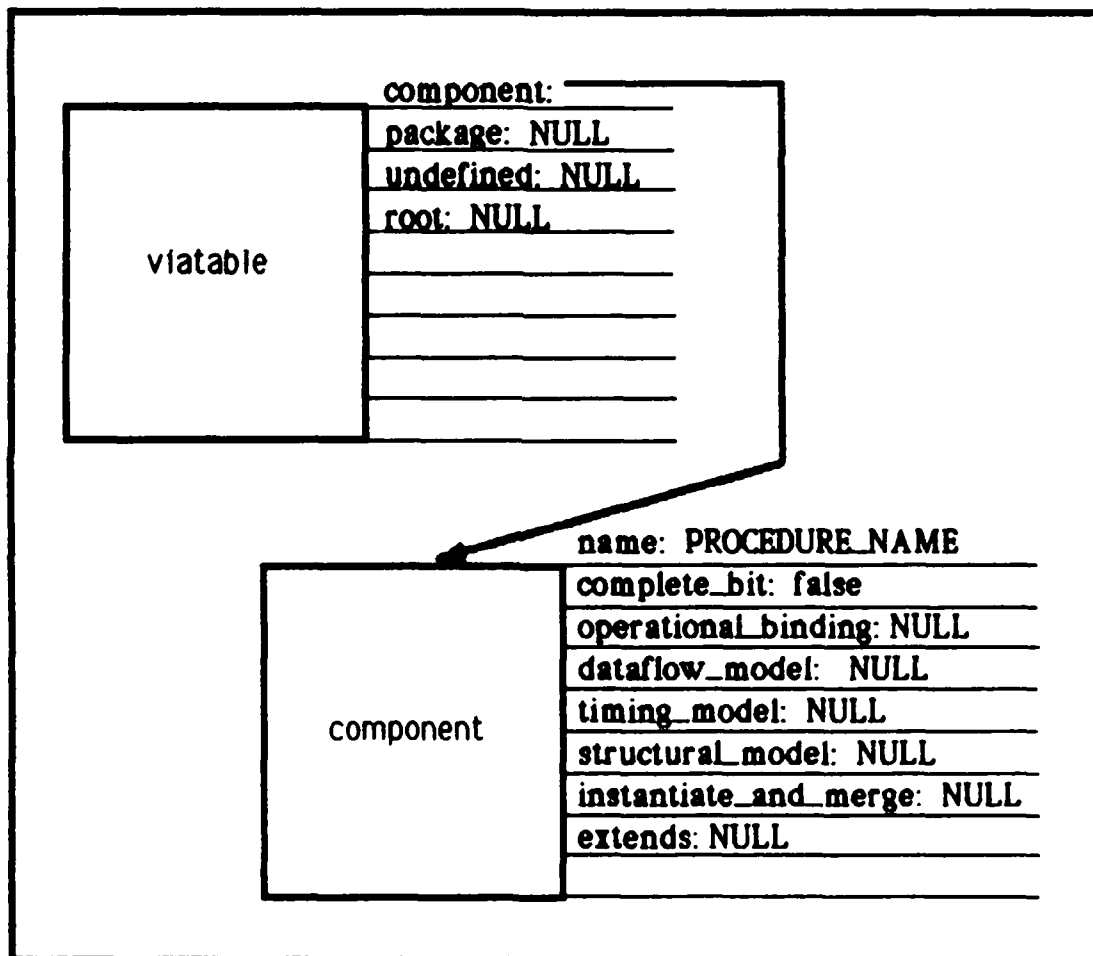
Functional tests. A representative set of functional test cases was created to verify that the language analyzer was consistent with the language reference manual (Intermetrics, 1985a). The functional test cases were similar in format, varying only with the exact input and expected output. Therefore, one specific example will be discussed, although the other tests appear in Appendix C. The particular example selected for discussion is depicted in Figure 5.1, "A Single Grammar Rule Test". For this test, the VHDL source code displayed at the top of the figure was the input data, while the VIA representation at the bottom of the figure was the expected result. To perform the test, the command

```
vhdl test_proc_1
```

was issued from the UNIX command line. *vhdl* is the name of the executable file for the language analyzer, and *test_proc_1* is the file containing the

VHDL Source Code:

```
procedure PROCEDURE_NAME is
begin
    null;
end ;
```

Enhanced DDS:**VIA Representation:**

```
0 via_table ( component = 1 ; )
1 component ( name = PROCEDURE_NAME ; complete_bit = false ; )
```

Figure 5.1: Procedure Test Case.

VHDL code displayed in Figure 5.1. The test creates three files (*viatable*, *viapack*, and *viatemp*), which when concatenated together, create the *via.vhdl* file discussed in Appendix B. The contents of the files were manually compared to the expected results as shown in the figure. In this case, the test results were exactly as depicted in Figure 5.1.

With some tests, however, the expected results varied from the actual results in one of three acceptable ways. First, the specific record numbers assigned to each record in the actual results might differ from those assigned in the expected results. Since record number assignments are arbitrary, provided the specific record numbers were unique, the test was deemed successful. Similarly, the order in which the records appeared was deemed insignificant, provided the record numbers were unique. Third, the actual results might reflect the contents of the expected results in more than one actual record through the use of the *extends* attribute. The *extends* attribute links two records together: a *primary record* (which can be any VIA record type) and an *extend record* (which is a record of the same type as the primary record). The extend record points to the primary record through the *extend* attribute. Basically, the extend record is a continuation (or extension) of the primary record. Therefore, the test was considered successful provided the union of the information in the actual results mapped into the contents of the expected results.

Performance tests Four sets of VHDL code were analyzed ten times by the language analyzer. These tests were performed under normal operating condition of the AFIT UNIX environment. To perform these tests a UNIX shell

was written to repetitively execute the statement

```
time vhd1 <filename>
```

The command *time* is a UNIX utility which reports elapsed time, user execution time, and system time (AT&T, 1986); *vhd1* is the name of the executable file for the language analyzer; and *filename* is the name of the file containing the VHDL source code. Table 5.1 reflects the raw data gathered from these tests, and Table 5.2 reflects the mean and standard deviation for the execution times. As the work progressed on the language analyzer, the mean execution time increased from 0.64 to 1.03 CPU seconds. Both the execution times and the output file sizes grew rapidly, because the third subset was the largest subset implemented. The first subset, the Design-File problem, processed six production rules (i.e., definitions YACC uses to create the parser); the second subset, the Context-File problem, processed approximately 20 production rules; and the third subset, the Declarations Problem, processed approximately 65 production rules. With these first three subsets very few of the production rules were shared. Those production rules which were used by more than one subset, were simple productions like:

```
simple_name := Identifier ;
```

whereas the third subset made a major contribution to future subset implementation. The third subset processed approximately one third of the 246 productions used by YACC. Since the third subset language analyzer performs over one-third of the expected total translation work, such a radical increase in time and file sizes is justified. The original performance projection for the completed language analyzer was to process a 1000-line VHDL source file within three CPU minutes. Although the test cases did not

Test Number	Input File Size	Subset Number							
		0	1	2	3	0	1	2	3
	(bytes/lines)	User Time (seconds)				Output File Size (bytes)			
A 1	2207/96	0.3	0.3	0.4	0.7	0	637	637	13532
2		0.3	0.3	0.4	0.7				
3		0.3	0.3	0.4	0.7				
4		0.3	0.3	0.3	0.7				
5		0.4	0.3	0.3	0.7				
6		0.3	0.4	0.3	0.7				
7		0.3	0.3	0.4	0.7				
8		0.3	0.4	0.4	0.7				
9		0.3	0.4	0.3	0.7				
10		0.3	0.3	0.4	0.8				
B 1	7978/293	1.4	1.4	1.4	2.1	0	1212	1208	18555
2		1.5	1.3	1.5	2.1				
3		1.5	1.4	1.4	2.0				
4		1.2	1.4	1.5	2.1				
5		1.5	1.4	1.5	2.0				
6		1.5	1.4	1.4	2.1				
7		1.4	1.4	1.4	2.0				
8		1.5	1.4	1.4	2.0				
9		1.5	1.5	1.4	2.2				
10		1.5	1.5	1.4	2.1				
C 1	3244/106	0.5	0.4	0.5	0.8	0	363	363	7542
2		0.4	0.5	0.5	0.7				
3		0.4	0.5	0.5	0.7				
4		0.5	0.5	0.5	0.7				
5		0.5	0.5	0.5	0.7				
6		0.4	0.3	0.5	0.7				
7		0.4	0.5	0.5	0.7				
8		0.4	0.4	0.5	0.8				
9		0.5	0.4	0.5	0.8				
10		0.4	0.5	0.5	0.7				
D 1	2392/83	0.4	0.4	0.4	0.6	0	311	311	7501
2		0.4	0.4	0.4	0.6				
3		0.4	0.4	0.4	0.6				
4		0.3	0.4	0.4	0.6				
5		0.3	0.4	0.3	0.6				
6		0.3	0.3	0.4	0.6				
7		0.4	0.4	0.4	0.6				
8		0.4	0.4	0.4	0.6				
9		0.3	0.4	0.3	0.6				
10		0.4	0.4	0.4	0.6				

Table 5.1: Performance Test Raw Data

Test Number	Subset Number							
	0	1	2	3	0	1	2	3
	Mean User Time seconds				Standard Deviation seconds			
A	0.31	0.33	0.36	0.70	0.03	0.05	0.05	0.00
B	1.45	1.41	1.43	2.07	0.09	0.05	0.05	0.06
C	0.44	0.46	0.50	0.73	0.05	0.07	0.00	0.05
D	0.36	0.39	0.38	0.60	0.05	0.03	0.04	0.00
	0.64	0.65	0.67	1.03	0.47	0.45	0.45	0.61

Table 5.2: Execution Time Means and Standard Deviations

conform to this criterion, the potential for the final prototype language analyzer to meet or exceed this criterion is good.

Stress tests. As stated earlier, stress tests were informal, and therefore, specific test cases were not created. Rather, two AFIT Hardware Architecture classes were assigned homework which required the students to write VHDL descriptions of various hardware configurations. During the winter quarter of 1986, students were assigned the task of describing a hardware system's port interconnections using configuration bodies (George, 1986). At the time of the assignment, the pre-prototype language analyzer was made available for student usage on the AFIT host UNIX system. The students were told the language analyzer processed VHDL syntax while ignoring VHDL's semantics. They were asked to report any problems which they discovered. All problems they found were attributed to learning a new language, and no problems associated with the language analyzer were reported. The author had the opportunity to review selected examples of the

students' code. Of the examples, few would have passed the semantic checks implemented in later releases of the analyzer.

During the summer quarter of 1986, students were assigned the task of describing the behavior of a Reduced Instruction Set Processor (RISC) they were designing as a class project (Linderman, 1986). This time the students used a release of the language analyzer which recognized VHDL the semantic shells. Once again no problems were reported. Yet this class assignment tested a different set of VHDL syntax and the analyzer's initial semantic capabilities.

Structural tests. As discussed earlier, structural tests were created for complex or critical program modules. One such group of program modules was the attribute modules which generate VIA. Basically, a program driver was written to test the attribute program modules; then the driver was executed using data which was selected to test every program statement. Most attribute program modules have two inputs (a symbol table pointer and an attribute table pointer). These programs call the symbol table routines to retrieve information from the symbol table which is printed during the creation of new VIA records. Also, these programs print the relationship of the new record to the partial directed acyclic graph (DAG) represented by the attribute table; then the new relationship is posted to the DAG.

The structural tests showed the modules worked correctly with one consistent exception. When both input parameters were null, the attribute programs printed null records (i.e., records which had no field names and no

field values). These null records were printed because the statement which checked this condition had been improperly formed.

Summary.

This chapter presented the specific test requirements, test methodology, representative test cases, and an evaluation of the test results. Four classes of tests were presented: functional, performance, stress, and structural. Of these four classes of tests, the functional and performance tests were primarily used for the analysis. The emphasis was placed upon functional testing throughout the incremental development, because the prototype analyzer's first objective was function as discussed in Chapter 1. The performance test showed that the language analyzer was processing an average of 4000 bytes (or 145 lines) of VHDL code in 1.03 CPU minutes. Since the language analyzer is currently processing over one-third of VHDL productions, the final prototype language analyzer can be expected to meet the initial performance objective of 1000 lines of VHDL code within three CPU minutes.

VI. Conclusions

Overview.

During this research, several conclusions were derived which not only apply to VHDL Version 7.2 (the version of VHDL for which the language analyzer was created), but also can be applied to the IEEE's enhanced VHDL described in Chapter 1. This chapter summarizes those conclusions and, based upon the conclusions, recommends six topics for future research endeavors.

Principal Conclusions.

The research approach stated in Chapter 1 called for the selection of a VHDL Intermediate Assess (VIA) format, selection of VHDL subsets, identification of logical relationships between VIA and VHDL, design of the language analyzer, and evaluation of the language analyzer. The latter three steps were applied iteratively on each subset throughout the project development cycle. Although in general this was a successful approach, minor setbacks were associated with the successes. These successes and setbacks are discussed in the following paragraphs.

1. Selection of VIA for the AFIT VHDL Environment. Selecting a structure for the intermediate files was the process of analyzing the VHDL language to determine a method for representing the content of a VHDL source program. As explained in chapter 3, Design Data Structure (DDS) as presented by Knapp and Parker of the University of Southern California (Knapp and Parker, 1984:

9-27) was selected as the basis for the underlying structure of the VIA format.

One reason for the selection of DDS was to allow the tools under parallel development to have early access to complete circuit descriptions. However, since the VIA definition was on the critical paths of the parallel projects, the scope of those projects was changed, so that the critical path did not depend upon the VIA definition. Nevertheless, VIA was considered for making the tool builders' respective design decisions.

2. Selection of the VHDL Subsets. Selection of the VHDL subsets to implement was the process of classifying the language rules into groups of related rules. The rule subsets were ranked in the order of expected implementation complexity. As each rule set was added to the analyzer, the analyzer's capabilities expanded. Computer code was designed and tested to validate the expanded capabilities.

This method of selecting subsets of the VHDL language was a success. It allowed early access to the intermediate files and enabled AFIT students to test the early analyzer releases as part of their normal homework assignments. In assuring that the early releases functioned properly, these students also gained a better understanding of both VHDL and the software verification process.

3. Identification of the Logical Relationship Between VIA and VHDL.

Identification of the logical relationship between the VIA and VHDL was the

process of determining how the semantics of the language are explicitly represented in the intermediate form. The language subsets were iteratively decomposed to provide examples of VHDL source code and the intermediate form. The examples served as a guide for designing modules and formed the test cases used for validation.

Although this task initially seemed relatively simple, as the design work progressed the task became complex and consumed more time than originally projected. The complexity of the task increased as more capabilities were added to the language analyzer, because VHDL and DDS have inherent differences which cause inconsistencies in the way they handle structure and behavior. For instance, VHDL's *types* can be assigned values through an *initialize* directive. DDS does not provide a consistent means of representing these initialize values for types, because VHDL's *types* are equivalent to DDS's *dataflow_link* records, which cannot acquire values. Dataflow_link records can point to *single_value* records, but the single_value records were used to represent the values associated with VHDL's *type* declarations (i.e., enumeration types and so forth). This inconsistency, like many others, was corrected by adding another field to the VIA file definition, and therefore, VIA is based upon enhanced DDS.

Due to similar complexities, the task of defining the logical VHDL-to-VIA relations required more time than originally expected. The task is still ongoing and is expected to continue as the prototype language analyzer expands into a production version based upon the IEEE's standardized VHDL.

4. Design Modules to Generate the Intermediate Structure. Designing modules to generate the intermediate structure was an iterative process of determining the actions required for generating the intermediate files. Modules were designed to meet the semantic criteria specified in the VHDL Language Reference Manual (Intermetrics, 1985a), to interface with YACC (Lesk and Schmidt, 1978), and to interface with a modified version of the Schreiner's and Friedman's symbol table (Schreiner and Friedman, 1985).

The design process in conjunction with the previous two steps required more time than was originally projected. Therefore, the scope of the project was re-evaluated. The goal of implementing all nine subsets within this project was changed to implementing three subsets. Although the goal changed, the incremental development methodology was a success. By developing the design in increments, the author was able to design and implement a set of robust program modules based upon abstract data types. These abstract data types are a set of data structures (i.e., symbol table, attribute table, and groups) with their corresponding operators (i.e., functions and procedures). The abstract data types allowed information hiding by separating low-level program details from high-level functional details and provided a large variety of functions and programs for implementing future VHDL subset capabilities.

5. Test and Evaluate the Language Analyzer. Testing and evaluating the language analyzer was the process of executing the VHDL language analyzer, checking the results against the predicted results, verifying the expected output, and determining run time performance. Four classes of tests were

performed: functional, performance, stress, and structural. Of these four classes, functional testing was emphasized, because the project goals emphasized function with performance as a secondary consideration. The performance tests indicated that as the development of the language analyzer continues the prototype analyzer should meet the projected goal of processing a 1000-line input file within three CPU minutes. Performance tests also showed that the output files are 2 to 6 times larger than the input files, and therefore, the record-names and field-names in the file should probably be shortened.

Suggestions for Future Research.

The current configuration of the VLSI tools at AFIT is an independent collection of automated tools which were developed at AFIT, acquired from other institutions, or purchased from industry. Among these tools are applications such as *SPICE* (Vladimirescu and others, 1983) (a gate circuit simulator), *CAESAR* (Ousterhout, 1983) (an interactive mask layout tool), *MEXTRA* (Fitzpatrick, 1983) (a circuit extractor), *LYRA* (Arnold, 1986) (a design rule checker), and many more. Although some of the tools such as *CAESAR* and *LYRA* communicate through a common data format, many of them require manual translation from one fixed format to another. This "man-in-the-loop" is notorious for making costly mistakes which are propagated through chip fabrication and discovered during chip testing. By removing the man-in-the-loop, whenever feasible, and by adding additional design tools to the AFIT VLSI design environment, many costly mistakes could be eliminated.

The AVE is potentially the kernel of an integrated VLSI design environment that would automate or make unnecessary such manual translations. This is so because VHDL is not only a simulation language, but also a hardware description language which can be used throughout the design cycle (from concept through testing). Also, the AVE must be highly integrated to become useful for training VLSI students and for developing VLSI chips. Currently, AVE is just another independent set of design tools (i.e., the language analyzer, the software simulator, etc.) Yet, given sufficient research time, the AVE could achieve full internal integration and applications (such as CAESAR and SPICE) could be integrated into AVE through VHDL. To achieve this result, the following research topics should be pursued.

1. Create a database manager with an associated database query language
2. Integrate CAESAR and SPICE into AVE.
3. Optimize the language analyzer's design and code.
4. Create a linker for the language analyzer.
5. Create an automatic VHDL code generator.
6. Create an automatic floor planner.

Each of these research topics is discussed below.

1. Create a Database Manager with an Associated Database Query Language.

A database manager with an associated database query language for the AVE environment would potentially solve two problems. First, a database manager would provide a means of controlling design files. Although the emphasis on such a database manager would be to control VHDL and VIA files, a secondary goal would be to support other files, such as CAESAR and SPICE files. Second, a database query language would eliminate the need for each tool in the AVE

to contain routines for traversing VIA's directed acyclic graph (DAG).

Although VIA was selected as an intermediate form because of its simple abstract subspaces, traversing the VIA DAG is a semi-complicated process which should be transparent to most tools. Additionally, the author speculates that many tools will not require information from all VIA subspaces; yet, without a database query language, the tools may be required to traverse the entire graph to acquire information from one particular subspace.

2. Integrate CAESAR and SPICE into AVE. Integrating CAESAR and SPICE into AVE through VIA would remove the "man-in-the-loop" discussed earlier. In current practice, VHDL simulation models are developed independently from SPICE's gate simulations and CAESAR's mask layouts. These products and/or applications describe different aspects of the same chip circuitry under development. In some cases, they even describe the same information from a different perspective. For example, CAESAR specifies the coordinates of various mask levels; from these coordinates, the length and width of the transistors formed by the mask levels can be determined. SPICE uses the length and width of transistors to perform gate-level simulations for deriving timing diagrams. The system requirements, such as timing, power, component hierarchy requirements, for a chip can be specified in VHDL. Although each of these applications require data in a specific format, a common data structure such as VIA were used as an intermediate form. Automated tools could be developed (1) to translate the VHDL to the timing aspects of SPICE, (2) to translate the mask levels of CAESAR to the physical aspects of SPICE, and (3) to translate the

NO-A178 648

AN IMPLEMENTATION OF A LANGUAGE ANALYZER FOR THE VERY
HIGH SPEED INTEGRAT. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..

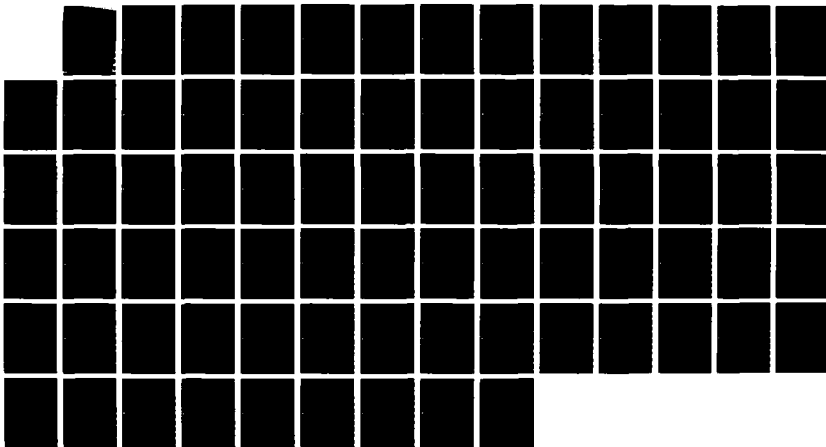
2/2

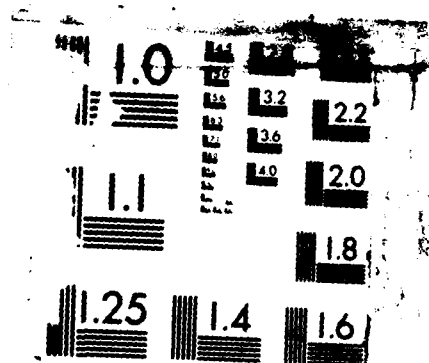
UNCLASSIFIED

D J FRAUENFELDER DEC 86 AFIT/GCE/NA/86D-1

F/G 9/2

NL





MIC

NA

translate the physical aspects of CAESAR to the timing aspects in SPICE. Admittedly, these translations are not simple and they may require human intervention in the form of on-line interactive sessions. Yet, even an on-line interactive session would reduce the potential of errors in omission, calculation, or transcription.

The current configuration of VIA supports dataflow, timing, and structure descriptions. Although the VIA does not currently support physical descriptions, the extension of VIA to include physical descriptions is easily accomplished by defining and adding the VIA record formats to describe the Design Data Structure (DDS) physical subspace. With respect to VHDL, there exists a direct correlation between the structural subspace and the physical subspace, so modifying the language analyzer to acknowledge the physical subspace would be relatively simple. Once the VIA physical subspace records were defined, tools could be written to translate the timing subspace to a SPICE shell and to translate the physical subspace to a CAESAR shell. SPICE and CAESAR require information not provided in VHDL. Therefore, to ensure that VIA contains complete descriptions, tools could be written to extract information from SPICE and CAESAR to store in VIA. The combination of tools would ensure that all known information on a given design was stored in one location, which would simplify the verification and validation process.

3. Optimize the Language Analyzer's Design and Code. As mentioned in Chapter 1, the primary goal of this research endeavor was to produce a functionally correct prototype language analyzer. Since this goal was accomplished, a logical continuation would be to optimize the performance of

the prototype analyzer. Three principal research goals could be (1) reduce the actual execution time from analyzing a single 1000-word source description in three CPU minutes on a VAX-class machine to 30 CPU seconds or less; (2) expand the language analyzer to allow multiple input files to be specified in the command line; and (3) expand the language analyzer to create multiple output files which correspond by name to the multiple input files. The optimization or near-optimization of the language analyzer would create a product which AFIT could be prouder to distribute among other universities.

4. Create a Linker for the Language Analyzer. A linker for the AVE environment is required to logically connect VIA files which were analyzed separately. The actual tasks associated with the linker would be similar to any other language linker, with the exception that the linker would be working with VIA files rather than machine language files. The linker would need to perform the instantiate-and-merge actions, related to the VIA files, which are described in Appendix B. Although the language analyzer currently requires an entire description in one file, the linker coupled with the previously discussed language analyzer enhancements would render a more usable language analyzer for the AVE.

5. Create an automatic VHDL code generator. The author's favorite recommendation for future research is an automated VHDL code generator. Like Ada, much of the code written for VHDL is to support user-defined data abstractions (i.e., data types and associated operations which can be performed using objects of those data types) and user-defined system design hierarchy. The nature of both data abstractions and design hierarchy renders

them potentially good applications for graphical definitions. Ideally, the VLSI designer would draw pictures (using rudimentary shapes such as boxes, triangle, circles, ovals, arrows, lines, labels, etc. selected from a menu), then these pictorial definitions would be translated into basic VHDL shells in which the designer would insert final program logic such as assignment statements. The specific information in these shells would vary among designs, but basically the structural information (such as that found in architectural bodies, configuration bodies, component instantiations, type definitions, and subprogram declarations) could be generated using the graphical information. The graphical information coupled with the VHDL source description would create the design documentation. Therefore, by drawing pictures (which the designer would most likely do anyway) on a screen, the designer would reduce the time needed to formulate design documentation and to write VHDL source descriptions.

6. Create an automatic floor planner. VHDL (and therefore VIA) contains sufficient hierarchy information to facilitate an automatic floor planner. The basic hierarchical structure of a VHDL source description could be used (at any phase of the design process) to evaluate and to create the basic floor plan for the chip under development. With auxiliary information such as size limitations, an automatic floor planner could heuristically generate a CAESAR file based upon the information maintained in the associated VIA file. The CAESAR file would contain basic cell definitions with no actual layer definitions. The automatic floor planner coupled with the automatic code generator would provide a complete set of source documentation for the initial design reviews.

Summary.

Although continued work on the language analyzer is required to encompass the entire VHDL language, the current prototype has already been used for teaching AFIT students about VHDL. The language analyzer is the first known C-based implementation which operates in a UNIX environment. Since the analyzer provides basic capabilities (such as the lexical analyzer, parser, symbol table, message handler and so forth), several organizations have already requested the source code from which they plan to build production models. Therefore, this research project provided the first step toward integrating the AFIT VLSI design environment and also design environments at other institutions. As integration work continues, AFIT students and faculty can provide valuable recommendations to the IEEE community as the IEEE enters the acceptance stage of standardizing VHDL as an industry-wide hardware description language.

Appendix A: Deviations from the VHDL LRM.

This appendix discusses the December 1986 language analyzer implementation with respect to deviations from the VHDL Language Reference Manual (LRM) (Intermetrics, 1985a). The prototype language analyzer's capabilities cover approximately one-third of the language as defined in the LRM. Therefore, to reduce redundant discussion, this appendix has been organized to follow the LRM chapters. When functions listed in chapters or major sections of the LRM were not implemented, the chapter/section reference is provided along with the projected VHDL subset to which the reference was scheduled. When the capabilities were partially implemented, the appropriate justification is provided.

Chapter 1. Design Entities.

Implemented except for port lists. These are scheduled for the next subset, *expressions*.

Chapter 2. Subprograms.

Implemented except for parameter lists. These are scheduled for the next subset, *expressions*.

Chapter 3. Packages.

Implemented.

Chapter 4. Types.

Implemented except for `secondary_unit` declarations. They are scheduled for the *expressions* subset.

Chapter 5. Declarations.

Implemented except for interface lists and port lists which are scheduled for the next subset, *expressions*.

Chapter 6. Specifications and Directives.

Implemented except for association lists and directives which are scheduled for the *expressions* subset.

Chapter 7. Names and Expressions.

Not implemented. Scheduled for the subset titled *expressions*. The only names implemented thus far are simple and indexed names.

Chapter 8. Statements.

Not implemented. Scheduled for the subsets titled *sequential statements* and *concurrent statements*.

Chapter 9. Scope and Visibility.

Implemented.

Chapter 10. Design Units and Their Analysis.

Implemented with three exceptions: revisions specifications are ignored; the package *standard* is a library unit; and design entities are analyzed in any order. These three aspects of VHDL are examples of implementation dependencies recorded in the LRM which are required to support the design library and the design library manager. The AFIT VHDL Environment has no design library or its manager; therefore, these

requirements were ignored for the prototype language analyzer. Additionally, ignoring the order of analysis requirement made a more flexible design environment.

Appendix A. Lexical Elements.

Implemented with two exceptions: reserved words must be lower case, and identifiers are case-sensitive. According to an LRM note, "In some attributes the identifier that appears after the apostrophe is identical to some reserved words" (Intermetrics, 1985a: A-10). To implement this capability either identifiers and reserved word needed to be case-sensitive or syntax error recovery was required. Since case-sensitivity was the easiest to implement, the author chose to differentiate the reserved words from identifiers by using lower-case reserved words and case-sensitive identifiers.

Appendix B. Predefined Language Environment.

B.1 Predefined Attributes.

Not implemented. Scheduled for the subset titled *concurrent statements*.

B.1 Predefined Types and Subtypes.

Implemented.

Appendix C. Syntax Summary.

Implemented without error recovery. Error recovery is a desirable quality for a production model language analyzer, yet for the prototype language analyzer the author felt the time should be spent on

implementing other aspects of the language analyzer. Additionally, according to the LRM, "If any error is detected while attempting to analyze a design unit, then the attempted analysis is rejected..." (Intermetrics, 1985a: 10-3).

Appendix D. Glossary.

Not applicable. The glossary contains definitions.

Appendix B: VIA

Overview.

To assist in the creation of future AFIT VHDL Environment (AVE) design tools, the VHDL Intermediate Access (VIA) file format is completely defined in this appendix. The file format is explained without justification of the rationale behind the decisions which led to the development of the VIA file structure: the reader interested in this rationale should see Chapters 1 through 4 of the thesis. Supporting examples are provided in Appendix C.

This appendix presents four categories of information: an overview of the VIA file structure, the component structure, the *viatable* structure, and detailed record definitions.

Overview of the VIA File Structure.

The VIA file structure is an alphanumeric pile format. The file consists of one control record, the *viatable*, followed by one or more VIA records. The format for all records is the same:

```
record-number record-type-name ( field-name-1 = field-value-1 ; ... ;  
    field-name-n = field-value-n ; )
```

Each record starts on a new line with a positive integer record-number. The record-number of the control record is always zero. The record-number is followed by a space, then the record-type-name. The record-type-name indicates the type of the record and establishes the valid field-names for

that record type. Following the record-type-name is a list of field-names and field-values separated by semicolons and enclosed within parentheses. Only those field-names are printed which have an established value that is not a default value.

The equal symbol follows the field-name and a field-value follows the equal symbol. Each field-name has at most one field-value. The field-value is followed by a semicolon. A semicolon followed by a closing parenthesis indicates the record is complete. Any white space within a record is a delimiter, unless the white space is enclosed in either single or double quotation marks. White space includes blanks, tabs, new lines, and so forth.

The Component Structure.

The graph in Figure B.1 represents the overall structure of the basic DAG associated with an enhanced DDS component. The graph is composed of labeled boxes, arrows, and labeled ovals. Each box in the graph represents a record in the VIA notation with the label inside the box being the record-type-name. Associated with each box are attributes which acquire values (see next section for valid attributes). In VIA notation the attributes are called field-names and the acquired values are called field-values. The boxes are interconnected by thin and wide arrows. These arrows imply relationships between the source (parent) and destination (child) boxes. The parent has an attribute whose value is the relationship. In VIA these relationships are represented by record numbers. A relationship implies two

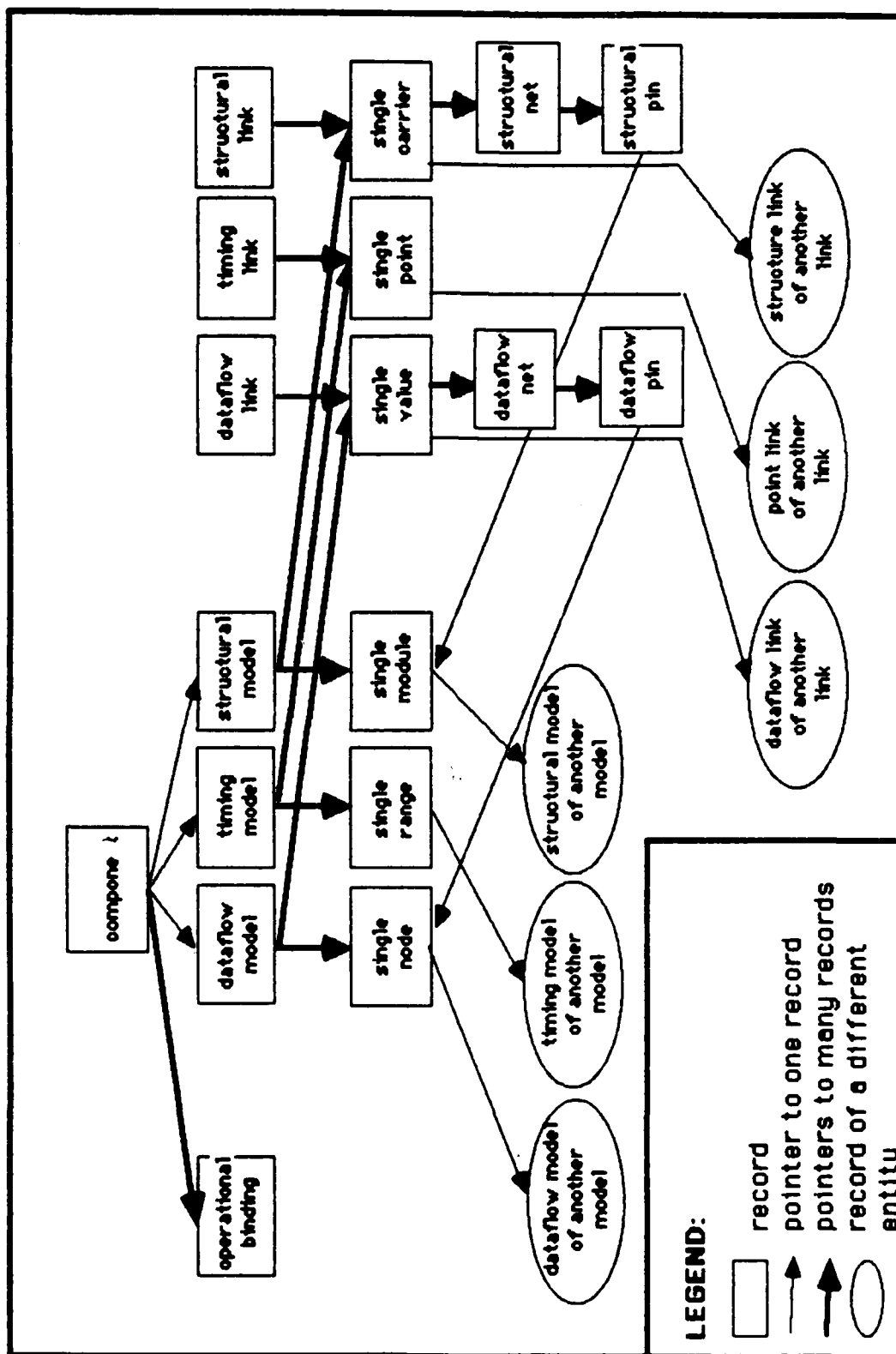


Figure B.1: VIA Record Hierarchy.

concepts: the child is actually part of the parent definition and the child inherits the attributes of the parent. A thin arrow implies the attribute will occur at most once, and a wide arrow implies the attribute can occur zero or more times. If an attribute of a parent has multiple relationships with a child, then a separate VIA record is created for each relationship. Although children inherit attributes from the parent, they do not inherit attributes from siblings (a parallel child). The arrow leading to labeled ovals represents a relationship with another component in exactly the same context.

The definition of each box with its associated attributes is discussed in the section labeled "Detailed Record Definitions", but first the structure of the viatable is presented.

The VIATABLE Structure.

As mentioned earlier, the *viatable* is the control record for the VIA file (see Figure B.2). In a VIA file, the viatable has the same structure as other records. As a control record, the viatable provides essential information for identifying top-level component records, package records, undefined symbol records, and a root record. Top-level component records are those component records which are derived from VHDL's architecture bodies, configuration bodies, and independent subprograms; package records are derived from VHDL's package declarations; undefined symbol records are derived from VHDL identifiers which were not declared; and the root record is the logical top of the DDS directed acyclic graph (DAG). With this information, design tools in the AVE can link between files and instantiate multiple copies of the

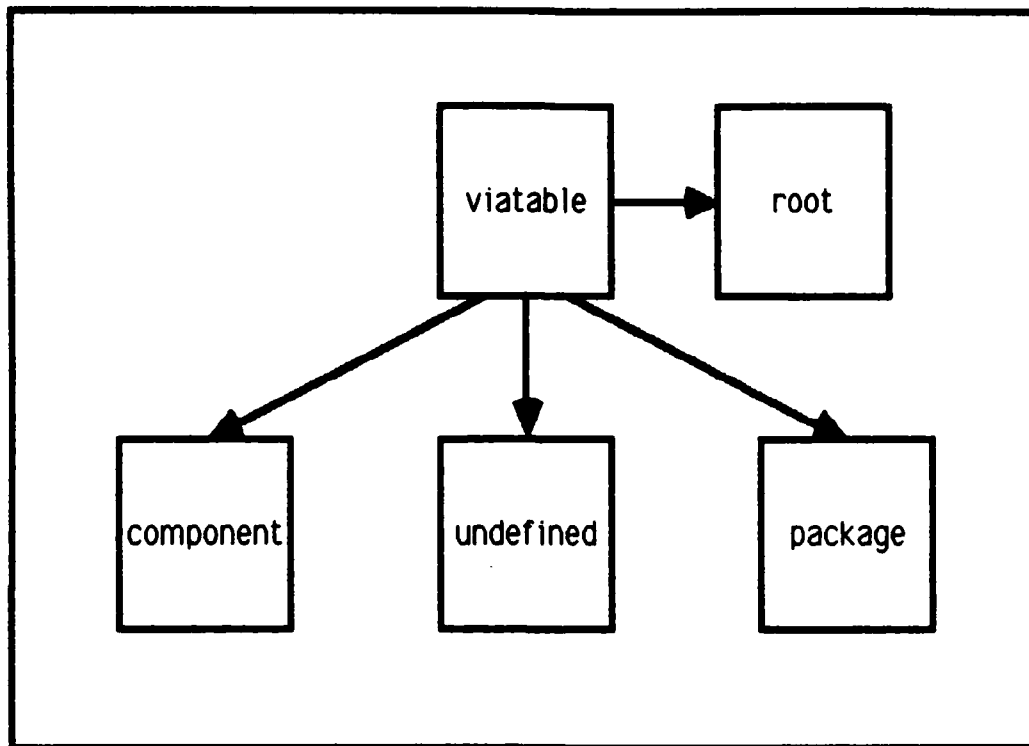


Figure B.2: VIATABLE Structure

enhanced DDS directed acyclic graph (DAG). To minimize memory requirements, only one DAG is created for any one VHDL construct. Yet, in many cases, VHDL requires instantiating copies of a particular VHDL design entity (Intermetrics, 1985a). Within the AVE, design tools have the responsibility for this instantiation. The necessary information required to instantiate copies of a particular DDS DAG (a secondary DAG) is embedded as `instantiate_and_merge` fields in that DAG (a primary DAG) which needs the copies. The `instantiate_and_merge` field provides the record number for the logical top of the secondary DAG. A unique copy of all records subordinate to the indicated record must be created, then the secondary DAG must be merged into the primary DAG. This merger implies two types of actions must occur. First, if both the primary and secondary DAGs begin with component records,

then a new component record must be created to represent the union of the two DAGs. The complete union includes merging the `dataflow_model`, `timing_model`, and `structural_model` records of the two component records to form new `dataflow_model`, `timing_model` and `structural_model` records. Second, all records which point to the original copy of the secondary DAG must be changed to point to the newly created DAG.

Any file which, in terms of DDS, describes a complete VHDL source description of an electronic component will have a field-name called `root`¹. The value associated with `root` is the record number of the component record which is the logical top of the entire DDS DAG. The root may instantiate any other component in the viatable, but it will never itself be instantiated in the resident file.

The linking between files and instantiation applies not only to the components which describe hardware but also to components which describe functions and procedures and to packages. The components which describe functions and procedures are handled exactly as previously described. Yet, to handle of components within a package, a linker or a design tool would need to know which components were within which package. Therefore, the record type *package* was created. The record type *package* contains a list of DDS components described in a VHDL package. Even this handles only half the problem; therefore, the record type *undefined* was created to describe undefined identifiers. The undefined record defines the context of an

1. If a linker is eventually created for AVE, then the linker has the responsibility of removing all but one root from the multiple files.

identifier which was used but not defined in a VHDL source description. When possible it also explains where the definition is expected to be found. For instance, it may state the identifier definition should be defined in a package and give the package name.

Detailed Record Definitions.

Each record which can appear in a VIA file is defined in this section. Each definition is composed of a record diagram, a record definition, and attribute definitions. The diagram shows the record-type-name inscribed in a box with a list of attributes extending to the right of the box. These attributes are the exact field-names which will appear in a VIA file. The attribute definitions explain the valid content of the field-values. At the risk of being redundant, each attribute for each record is defined with that record because the attribute definitions vary slightly among the various records.

1. **component Record:**

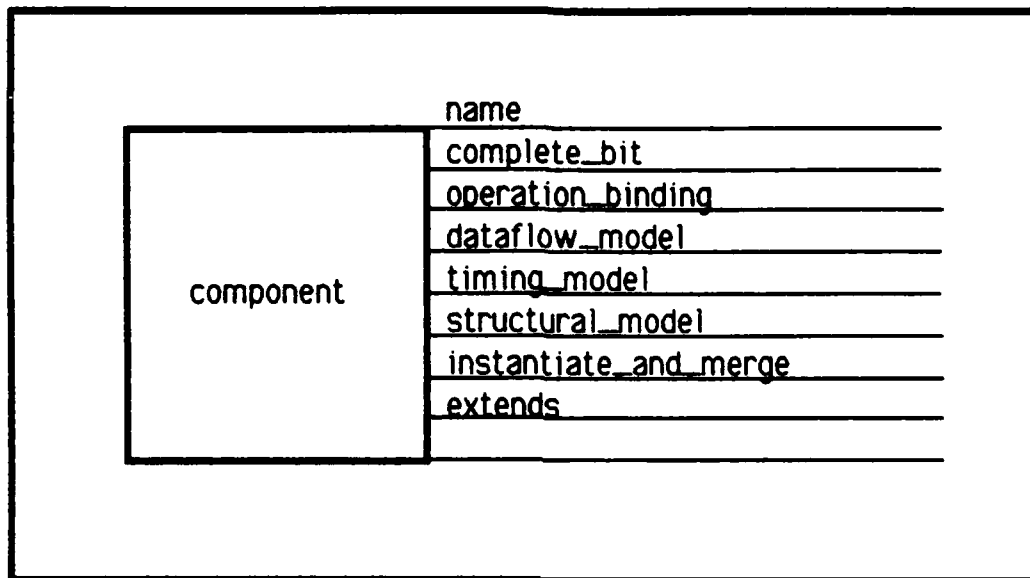


Figure B.3: component Record.

Record Definition: The component record is the top level record which explains the construction of a hardware or a software entity. The attributes point to the subspaces which describe the component through lower level subspace definitions.

Attribute Definitions:

- a. name is the name of the component.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

b. complete_bit indicates whether or not the VHDL description of the component is complete. The complete_bit attribute will appear in the component record whenever the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

c. structural_model is a record number for the structural_model record which describes the structural subspace of the component record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

d. timing_model is a record number for the timing_model record which describes the timing subspace of the component record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

e. dataflow_model is a record number for the dataflow_model record which describes the dataflow subspace of the component record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

f. operational_binding is a record number for an operational_binding record which describes a specific carrier-range-value or module-range-node relationship for the component record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

g. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

h. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

2. dataflow_link Record:

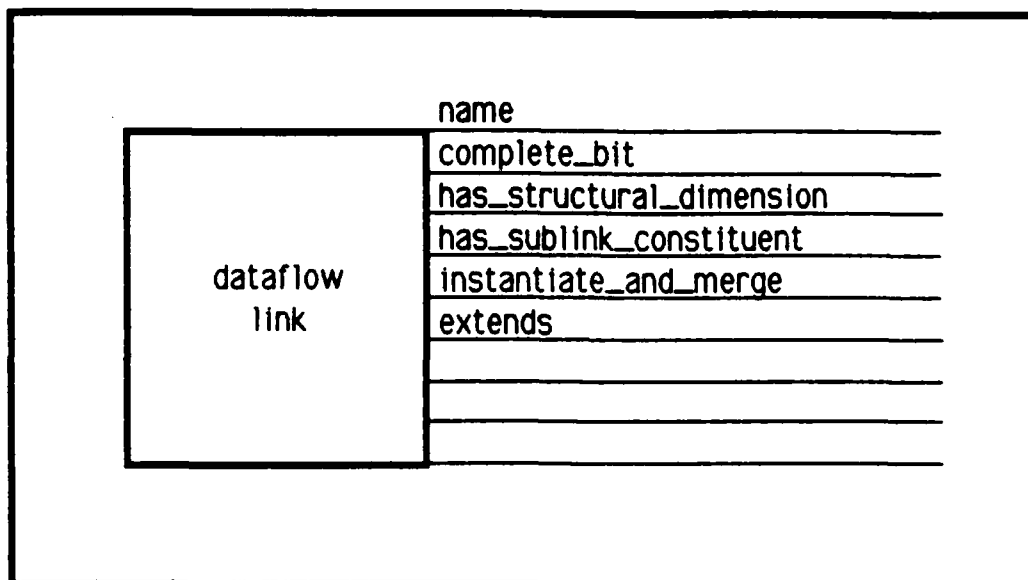


Figure B.4: dataflow_link Record.

Record Definition: The dataflow_link record characterizes the shared value dependencies between (1) a value and a node of the same component, and (2) values of two different components.

Attribute Definitions:

- a. name is the name of the dataflow_link.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. complete_bit indicates whether or not the description of the dataflow_link is complete. The complete_bit attribute will appear in the

dataflow_link record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

c. has_structural_dimension is an integer indicating the size of the dataflow_link.

Syntax: positive integer.

Occurs: at most once.

Default: 0.

d. has_sublink_constituent contains a record number for a single_value record which describes a value associated with a dataflow_link.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

e. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

f. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

3. **dataflow_model** Record:

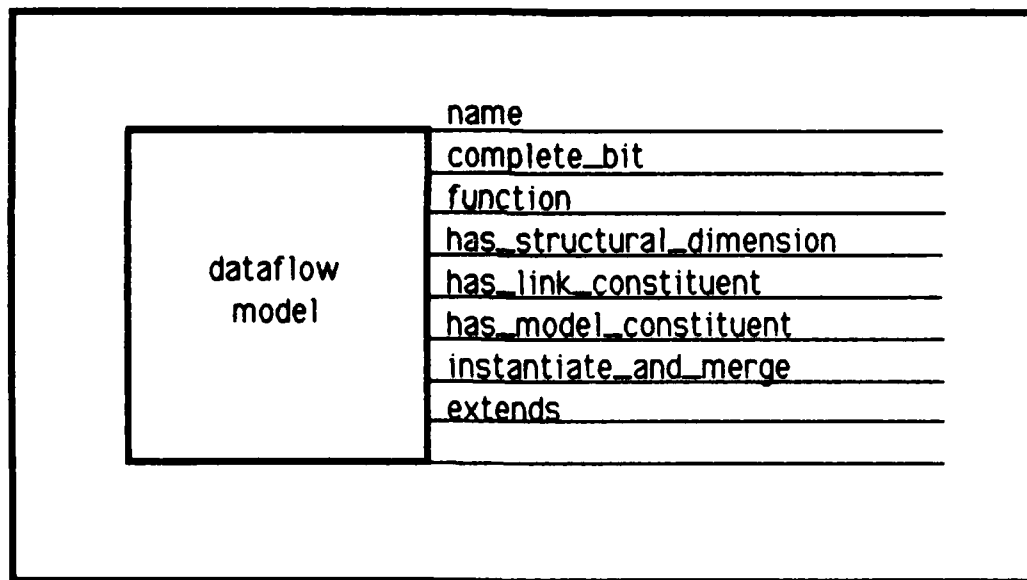


Figure B.5: dataflow_model Record.

Record Definition: The dataflow_model record is one of three subspace records; it characterizes the behavior of a component. The dataflow_model has two types of attributes which are explained at a lower level in the hierarchy: nodes and values. Nodes represent a functional transformation,

while values represent the initial conditions and/or the results of a functional transformation.

Attribute Definitions:

- a. name is the name of the dataflow_model.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. complete_bit indicates whether or not the description of the dataflow_model is complete. The complete_bit attribute will appear in the dataflow_model record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

- c. function indicates the purpose of the dataflow_model.

Syntax: character string.

Occurs: at most once.

Default: NULL.

- d. has_structural_dimension is an integer indicating the size of the dataflow_model.

Syntax: positive integer.

Occurs: at most once.

Default: 0.

- e. has_link_constituent contains a record number for a single_value record which describe a value associated with the dataflow_model.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- f. has_model_constituent contains a record number for a single_node record which describes a node associated with the dataflow_model.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- g. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- h. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

4. **dataflow_net** Record:

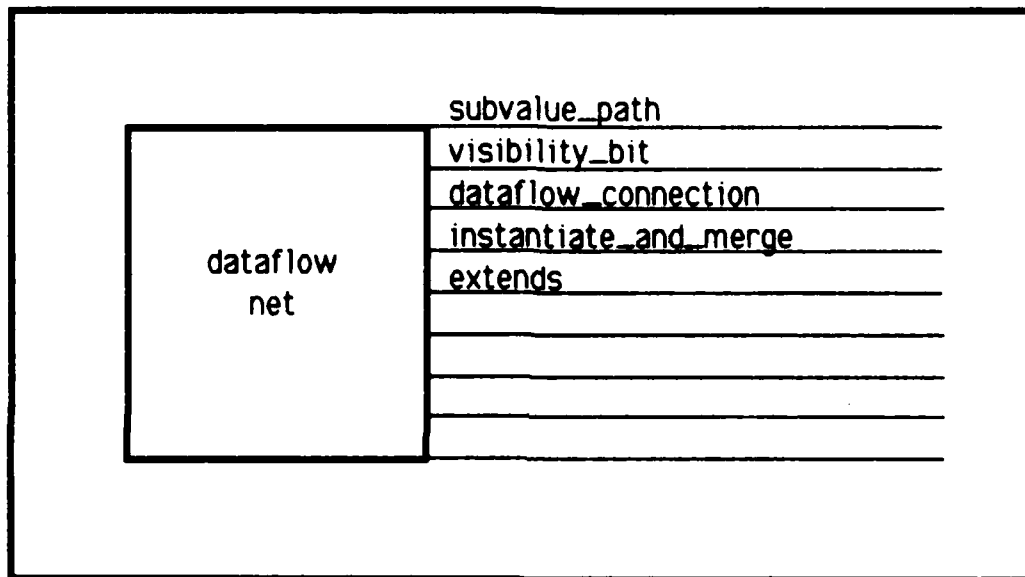


Figure B.6: dataflow_net Record.

Record Definition: The dataflow_net record binds together values of two different component.

Attribute Definitions:

- a. subvalue_path is a record number for the record which describes one aspect of a lower level decomposition of the single_value record which is the parent of the dataflow_net record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- b. visibility_bit defines whether or not the values can be accessed by a record in another subspace.

Syntax: true or false.

Occurs: at most once.

Default: false.

- c. dataflow_connection contains a record number for a dataflow_pin record which describes the interconnections for the dataflow network.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- d. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- e. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the

extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

5. **dataflow_pin Record:**

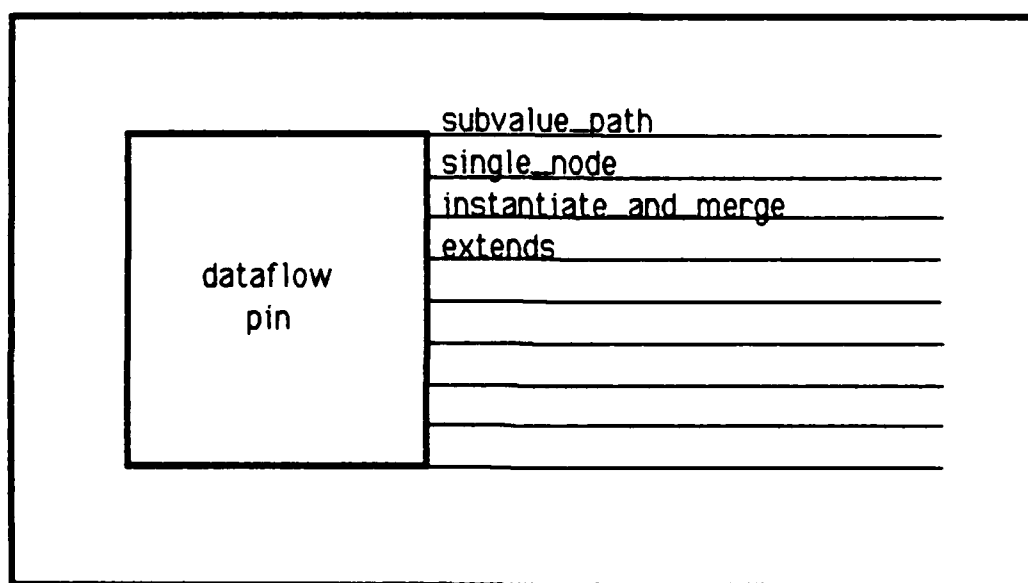


Figure B.7: `dataflow_pin` Record.

Record Definition: The `dataflow_pin` record binds together a value and a node of the same component.

Attribute Definitions:

- a. `subvalue_path` is a record number for the `single_value` record which is connected to the `single_node`.

Syntax: positive integer or NULL

Occurs: at most once.

Default: NULL.

- b. single_node contains a record number for the single_node record which is connected to the single_value record listed in the subvalue_path attribute.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- c. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- d. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

6. operational_binding Record:

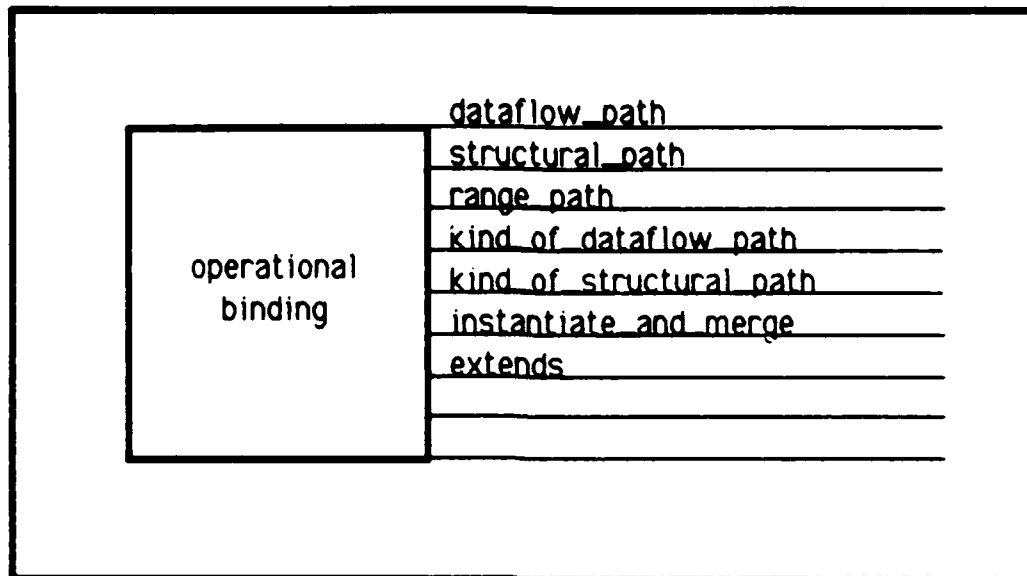


Figure B.8: Operational Binding Record.

Record Definition: "operational_binding [records] show the relationship between an operation (or value), a structure, and a time interval" (Afsarmanesh and others, 1985:31). Operational_binding records actually point to carrier_value_range and module_node_range bindings discussed by Afsarmanesh and others.

Attribute Definitions:

a. dataflow_path is a record number for either a single_value or single_node record. The specific record type is identified by the kind_of_dataflow_path attribute.

Syntax: positive integer or NULL

Occurs: once.

Default: NULL.

- b. structural_path is a record number for either a single_carrier or single_module record. The specific record type is identified by the kind_of_structural_path attribute.

Syntax: positive integer or NULL.

Occurs: once.

Default: NULL.

- c. range_path is a record number for a single_range record.

Syntax: positive integer or NULL.

Occurs: once.

Default: NULL.

- d. kind_of_dataflow_path represents the type of the dataflow path.

Syntax: node or value.

Occurs: once.

Default: no default.

- e. kind_of_structural_path represents the type of the structure path.

Syntax: module or carrier.

Occurs: once.

Default: no default.

- f. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record

needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

g. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

7. **package** Record:

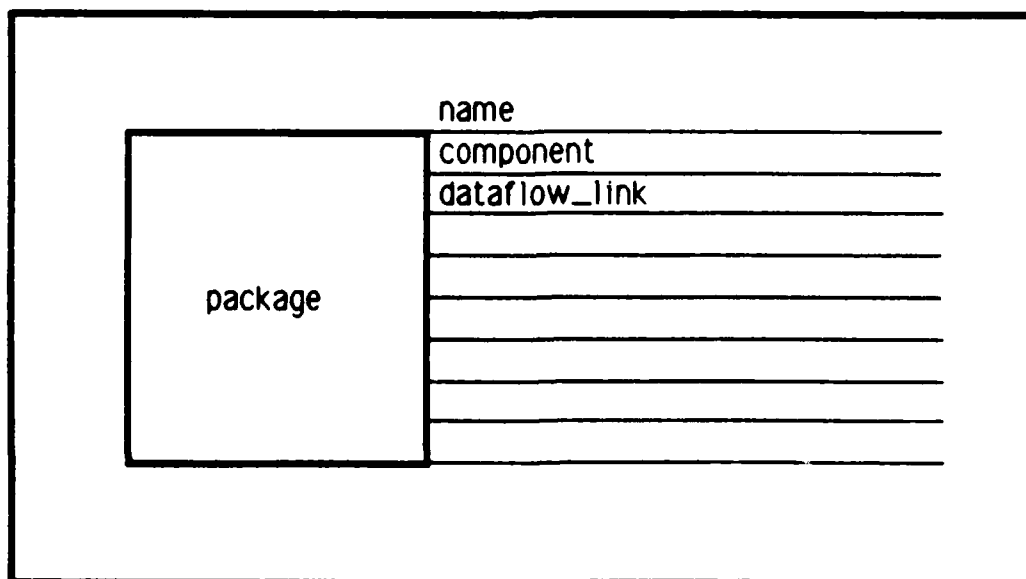


Figure B.9: Package Record.

Record Definition: The package record defines all lower level components and dataflow_links which were specified within a VHDL package.

Attribute Definitions:

- a. name is the name of the package.

Syntax: any valid VHDL package name.

Occurs: once.

Default: no default; all VHDL packages have names.

- b. component is a record number for a component record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- c. dataflow_link is a record number for a dataflow_link record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

8. single_carrier Record:

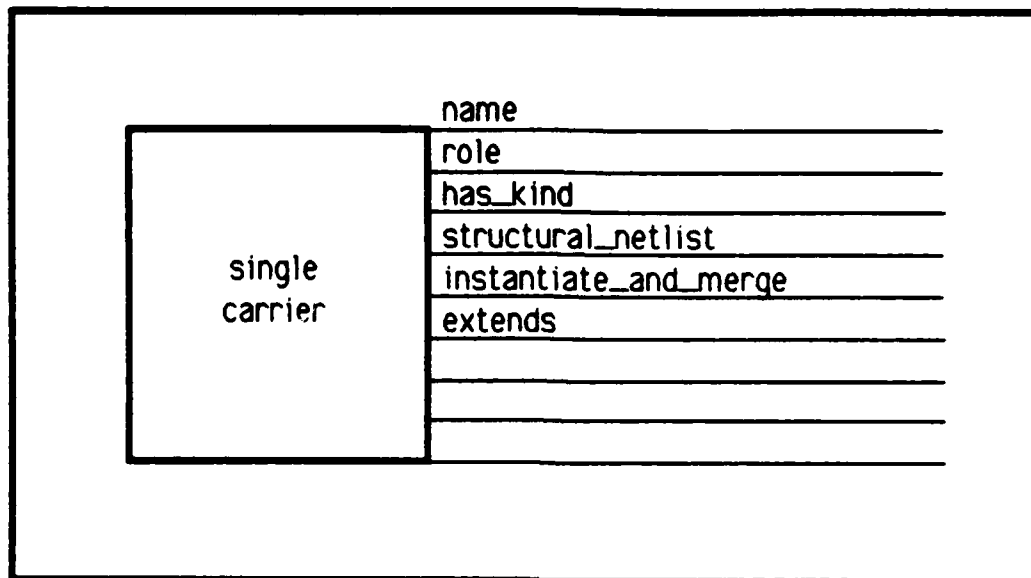


Figure B.10: Single_Carrier Record.

Record Definition: The single_carrier record characterizes the path for the results of a functional transformation.

Attribute Definitions:

- a. name is the name of the single_carrier.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. role is an integer which represents the reference number for VHDL arrays.

Syntax: integer or NULL.

Occurs: at most once.

Default: NULL.

- c. has_kind contains a record number for a structural_link record which is described by a different component record (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- d. structural_netlist contains a record number for a structural_net record which describes the network for the single_carrier.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- e. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- f. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information

listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

9. **single_module Record:**

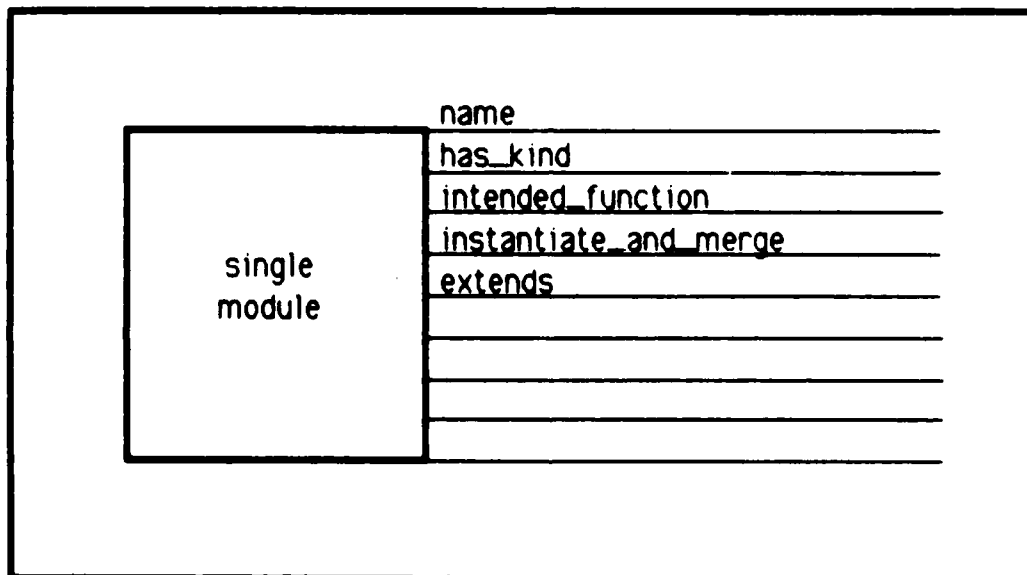


Figure B.11. Single_Module Record.

Record Definition: The `single_module` record characterizes a location for which a functional transformation occurs.

Attribute Definitions:

a name is the name of the `single_module`

Syntax any valid VHDL identifier

Occurs: at most once.

Default: NULL.

- b. has_kind contains a record number for a structural_model record which is described by a different component record (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- c. intended_function signifies the function performed "in the target design as opposed to the function ... as an isolated entity" (Afsarmanesh and others, 1985: 42).

Syntax: character string.

Occurs: at most once.

Default: NULL.

- d. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- e. extends is a record number for a previously printed record. Any record

which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

10. **single_node** Record:

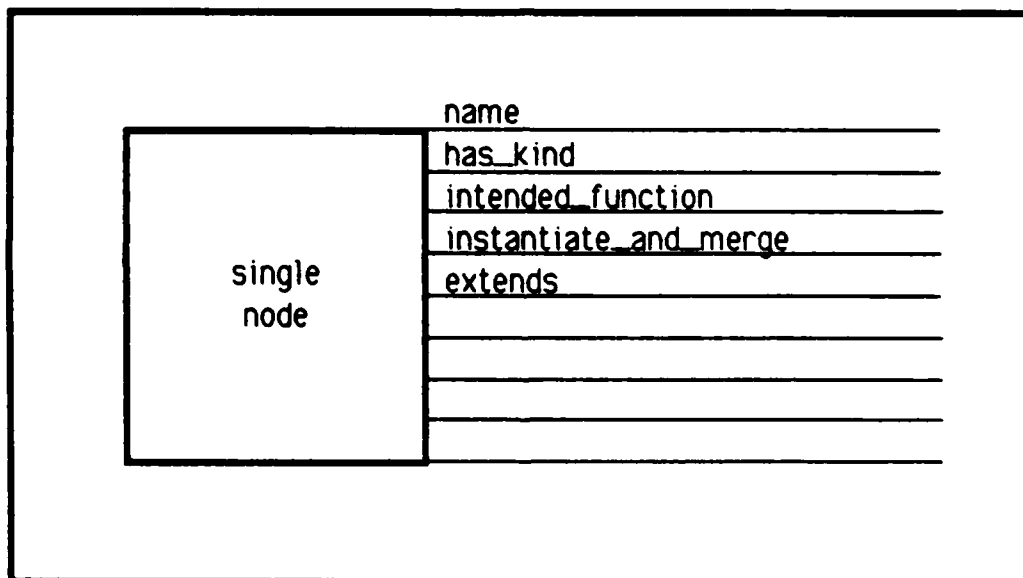


Figure B.12: Single_Node Record.

Record Definition: The single_node record characterizes a functional transformation of a component's dataflow subspace.

Attribute Definitions:

- a. name is the name of the single_node.

Syntax: any valid VHDL function name, procedure name, attribute name, or operator.

Occurs: once.

Default: no default.

b. has_kind contains a record number for a dataflow_model record which is described by a different component record (Afsarmanesh and others, 1985:42).

Syntax: positive integer or NULL

Occurs: at most once.

Default: NULL.

c. intended_function signifies the function performed "in the target design as opposed to the function ... as an isolated entity" (Afsarmanesh and others, 1985: 42).

Syntax: character string or NULL.

Occurs: at most once.

Default: NULL.

d. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

e. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

11. single_point Record:

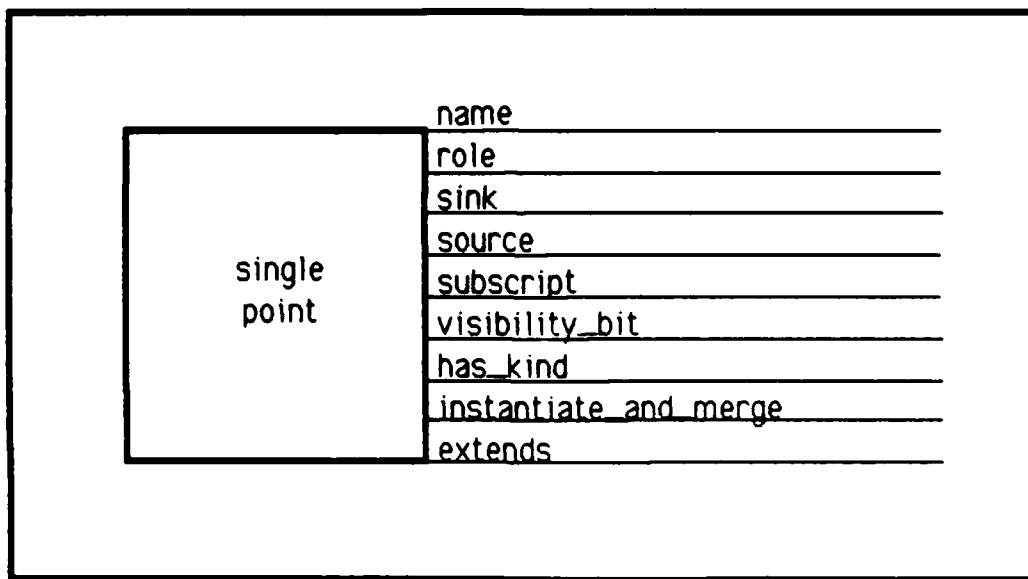


Figure B.13: Single_Point Record.

Record Definition: The single_point record characterizes the a specific time an event or a functional transformation occurs.

Attribute Definitions:

- a. name is the name of the single_point.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. role indicates the relative position of the single_point with respect to the start of a single_range.

Syntax: integer.

Occurs: at most once.

Default: zero.

- c. sink is a record number for the last single_range record which describes the connectivity of the timing diagram.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- d. source is a record number for the first single_range record which describes the connectivity of the timing diagram.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- e. subscript is an number representing a specific iteration of a loop.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL

f. visibility_bit defines whether or not the single_point record can be seen directly.

Syntax: true or false

Occurs: at most once.

Default: false.

g. has_kind contains a record number for a timing_link record which is described by a different component record (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

h. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

i. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the

extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

12. **single_range** Record:

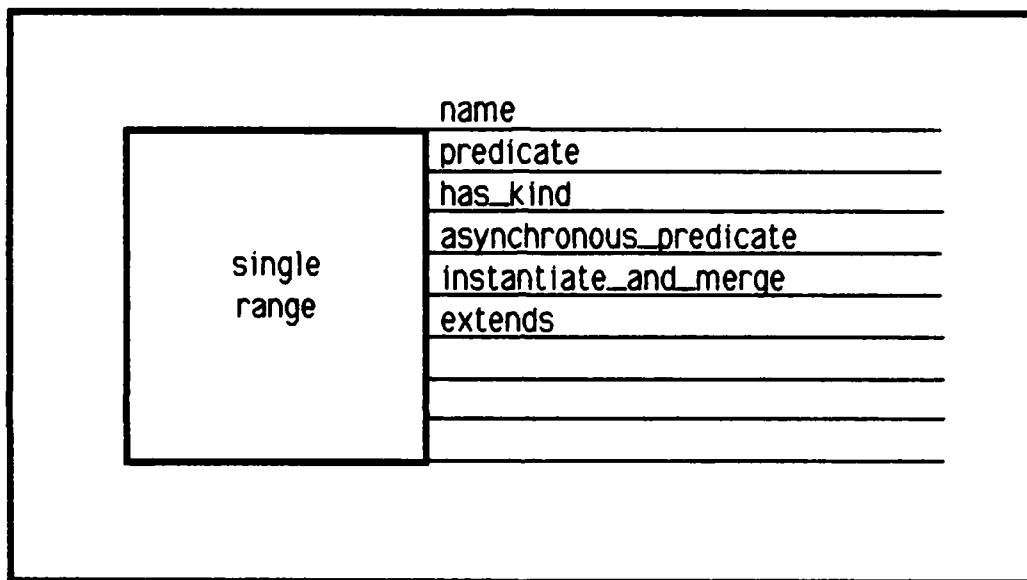


Figure B.14: Single_Range Record.

Record Definition: The single_range record characterizes a timing range of a component's timing subspace.

Attribute Definitions:

- a. name is the name of the `single_range`.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

b. "predicates describes the conditions under which normal branching will occur" (Afsarmanesh and others, 1985: 42-44).

Syntax: simple, alpha, omega, or_fork, and_fork, or and_join.

1) "simple points have one in-arc and one out-arc. These points represent events."

2) "alpha points have one out-arc and no in-arcs. These points represent loop re-entry points. The out-arc must have an index subscription as the loop is considered to be a (possibly infinite) set of instantiations of the arc(s) between alpha and omega points."

3) "omega points have one in-arc and no out-arcs. The points represent loop back jump points."

4) "or_fork points have one in-arc and a number of out-arcs. They represent branch points. Each out-arc must have a predicate attached to it describing the condition under which the arc is taken."

5) "and_fork points have a number of in-arcs and a single out-arc. They represent points at which several disjoint executions paths merge."

6) "and_join points have a number of in-arcs and a single out-arc. They represent co-end points."

Occurs: once.

Default: simple.

c. has_kind contains a record number for a timing_model record which is described by a different component model (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL

Occurs: at most once.

Default: NULL

- d. "asynchronous predicates" describes the conditions under which branching is not synchronized to a particular point in the time graph. (e.g., resets)" (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- e. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- f. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

13. single_value Record:

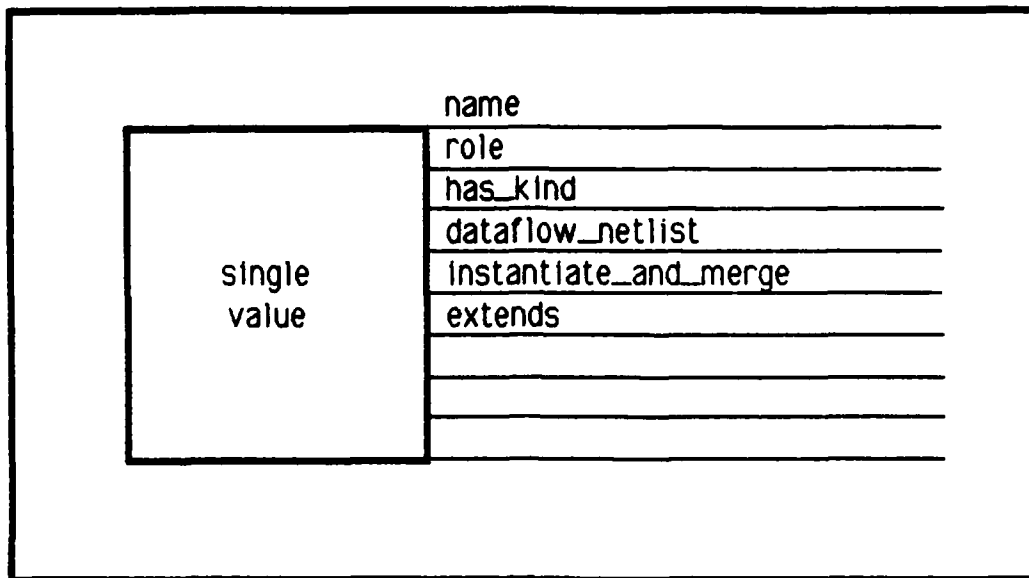


Figure B.15: Single_Value Record.

Record Definition: The single_value record characterizes either the initial conditions or the results of a functional transformation of a component's dataflow subspace.

Attribute Definitions:

- a. name is the name of the single_value.

Syntax: any valid VHDL identifier or constant.

Occurs: at most once.

Default: NULL.

- b. role is the index into a set of values which describe a dataflow_link.

Syntax: integer or NULL.

Occurs: at most once.

Default: NULL.

- c. has_kind contains a record number for a dataflow_link record which is described by a different component model (Afsarmanesh and others, 1985: 42).

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- d. dataflow_netlist contains a record number for a dataflow_net record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- e. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- f. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information

listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

14. structural_link Record:

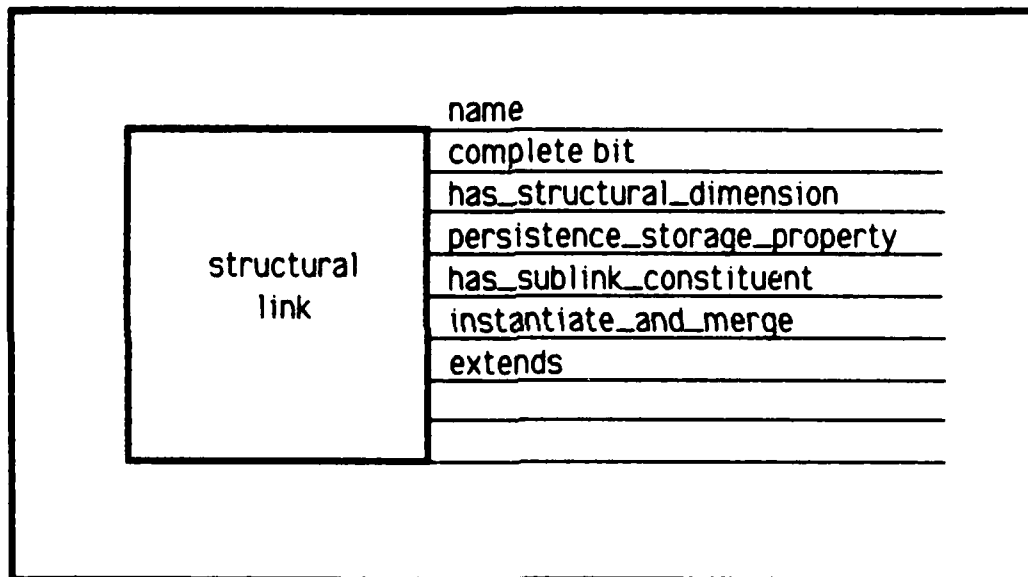


Figure B.16: Structural_link Record.

Record Definition: The structural_link record characterizes the shared carrier dependencies between 1) a carrier and a module of the same component and 2) carriers of two different components. Essentially, the structural_link is a binding.

Attribute Definitions:

- a. name is the name of the structural_link.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. complete_bit indicates whether or not the description of the structural_link is complete. The complete_bit attribute will appear in the structural_link record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

- c. has_structural_dimension is an integer indicating the size of the structural_link.

Syntax: integer.

Occurs: at most once.

Default: 0.

- d. persistence_storage_property describes the ability to store charge. Under some circumstances charge storage can be used as a memory mechanism.

Syntax: true or false.

Occurs: at most once.

Default: false.

e. has_sublink_constituent contains a record number for a single_carrier record which is described by a different structural_link.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

f. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

g. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

15. **structural_model Record:**

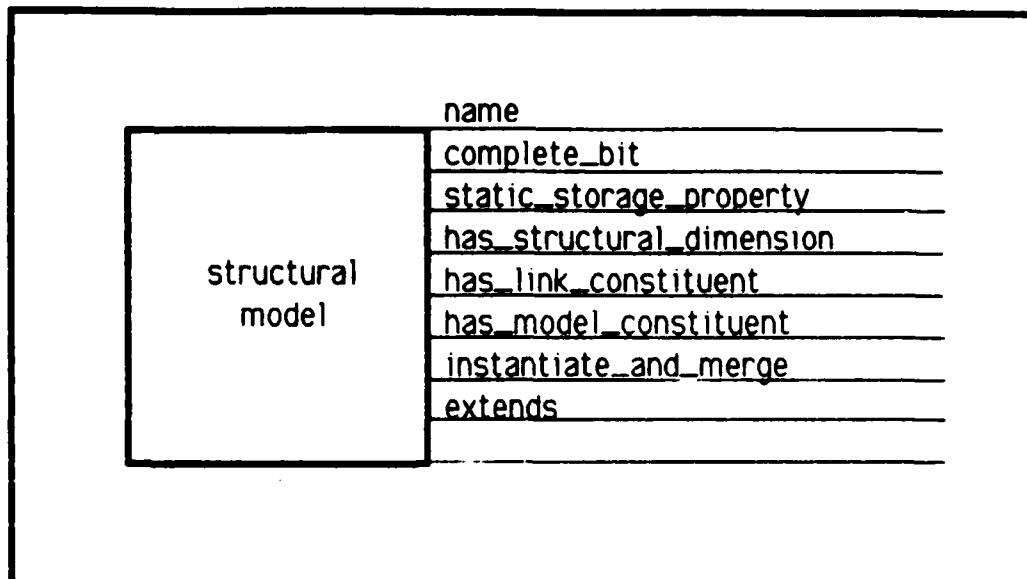


Figure B.17: `Structural_model` Record.

Record Definition: The `structural_model` record is one of three subspace records; it characterizes the structure of a component. The `structural_model` has two types of attributes which are explained at a lower level in the hierarchy: modules and carriers. A module is similar to a block on a schematic diagram indicating where a functional transformation occurs, while a carrier is similar to a line on a schematic diagram indicating the path for the results of a functional transformation.

Attribute Definitions:

- a. name is the name of the `structural_model`.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

b. complete_bit indicates whether or not the description of the structural_model is complete. The complete_bit attribute will appear in the structural_model record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

c. static_storage_property represents the modules ability to store a static charge (i.e., registers). The model is assumed to not store a charge unless the attribute is present in the structural_model record.

Syntax: true or false.

Occurs: at most once.

Default: false.

d. has_structural_dimension is an integer indicating the size of the structural_model.

Syntax: integer.

Occurs: at most once.

Default: 0.

e. has_link_constituent contains a record number for a single_carrier record which is described by a structural_model.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

f. has_model_constituent contains a record number for a single_module record which is described by a structural_model.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

g. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

h. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

16. structural_net Record:

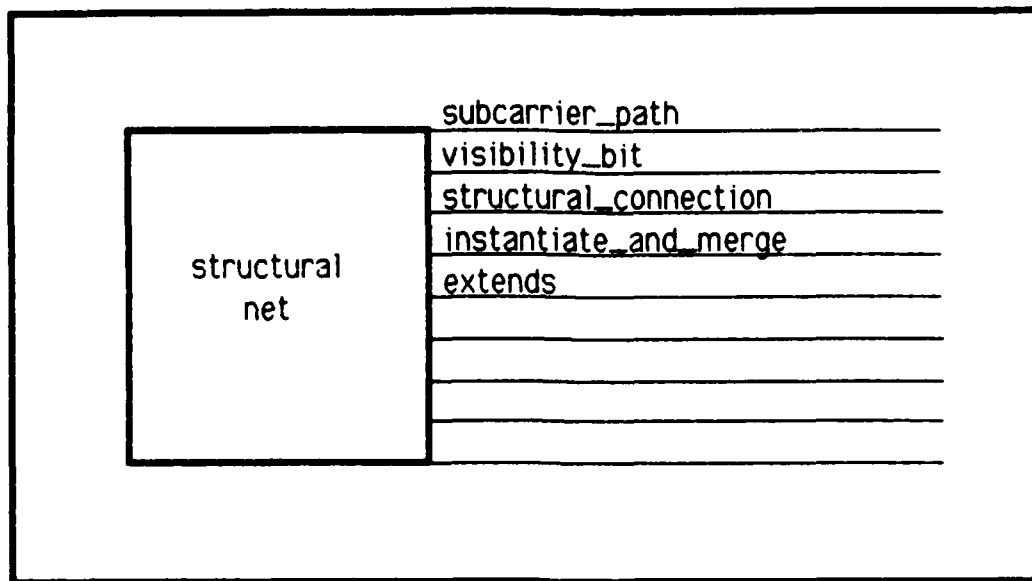


Figure B.18: structural_net Record.

Record Definition: The structural_net record binds together carriers of two different component.

Attribute Definitions:

a. subcarrier_path is a record number for a lower level single carrier record which describes the decomposition of the parent to the structural_net record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

b. visibility_bit is a flag stating whether or not the structural_net can be accessed.

Syntax: true or false.

Occurs: at most once.

Default: true.

- c. structural_connection contains a record number for a record which describes the structural connections for the structural network.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- d. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- e. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

17. structural_pin Record:

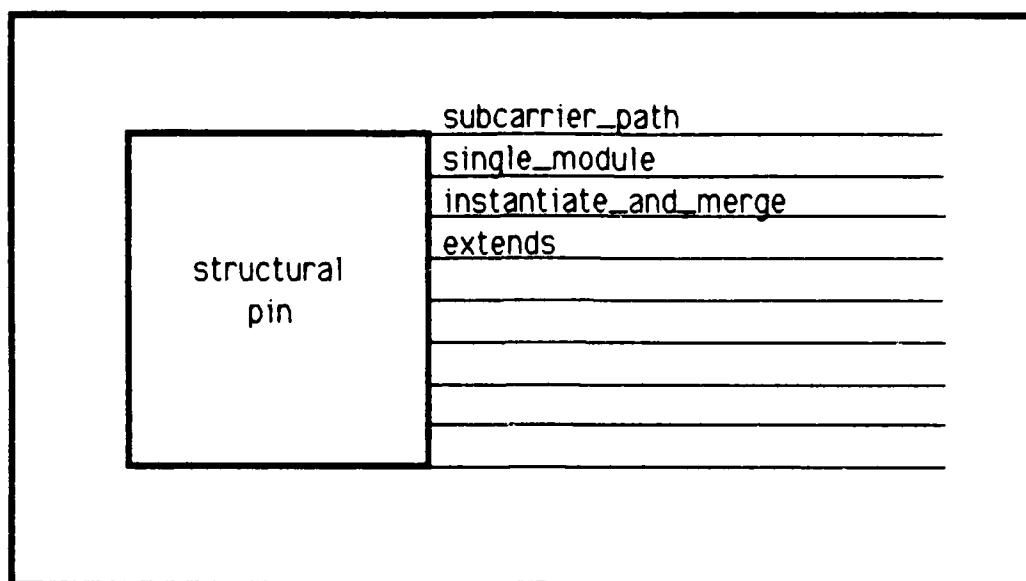


Figure B.19: structural_pin Record.

Record Definition: The structural_pin record binds together a carrier and a single_module of the same component.

Attribute Definitions:

- a. subcarrier_path is a record number for the single_carrier associated with the single_module.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- b. single_module contains a record number for a single_module record

Syntax: positive integer or NULL

Occurs: zero or more times.

Default: NULL.

c. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

d. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

18. timing_link Record:

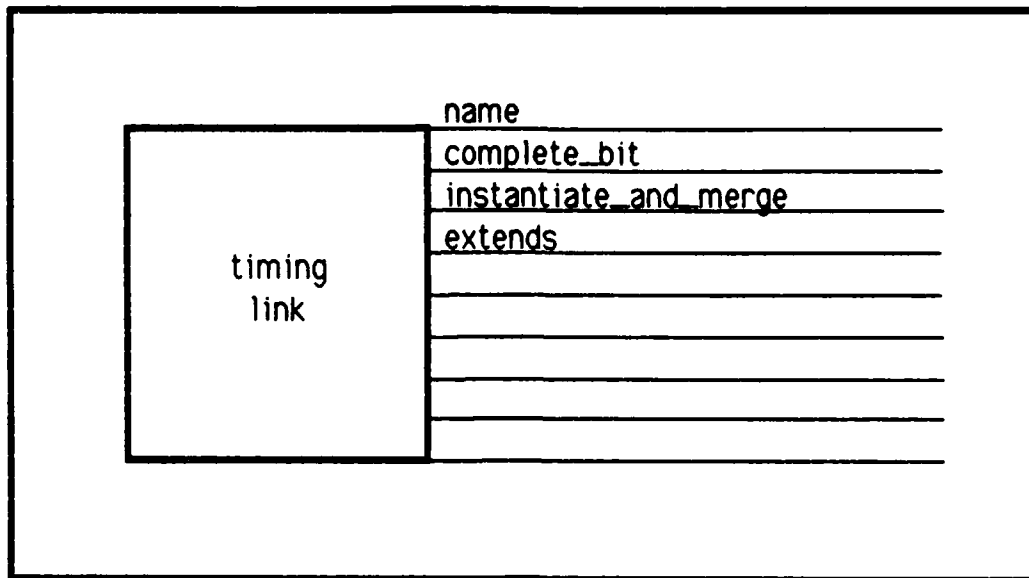


Figure B.20: Timing_link Record.

Record Definition: The timing_link record characterizes the shared point (i.e., an instance of time) dependencies between points of two different components. Essentially, the timing_link is a binding.

Attribute Definitions:

- a. name is the name of the timing_link.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

- b. complete_bit indicates whether or not the description of the timing_link is complete. The complete_bit attribute will appear in the

timing_link record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

c. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

d. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

19. timing_model Record:

timing model	name
	complete_bit
	duration
	causality
	has_structural_dimension
	has_link_constituent
	has_model_constituent
	instantiate_and_merge
	extends

Figure B.21: Timing_model Record.

Record Definition: The timing_model record is one of three subspace records; it characterizes the timing and sequencing dependencies of a component. The timing_model has two types of attributes which are explained at a lower level in the hierarchy: ranges and points. A range represents the time duration over which a functional transformation occurs, while a point is a specific time which an event will occur.

Attribute Definitions:

- a. name is the name of the timing_model.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

b. complete_bit indicates whether or not the description of the timing_model is complete. The complete_bit attribute will appear in the timing_model record when the description is not complete. Otherwise, the description is assumed to be complete.

Syntax: true or false.

Occurs: at most once.

Default: true.

c. duration indicates the length of the time interval.

Syntax: integer.

Occurs: at most once.

Default: 0.

d. causality indicates what caused the timing model record to be created.

Syntax: character string or NULL.

Occurs: at most once.

Default: NULL.

e. has_structural_dimension is an integer indicating the size of the timing_model.

Syntax: positive integer.

Occurs: at most once.

Default: 0.

f. has_link_constituent contains a record number for a record which describes the single_points which make up the timing_model.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- g. has_model_constituents contains a record number for a single_range record which is part of a timing_model.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

- h. instantiate_and_merge is a record number for the logical top of a previously printed DAG (i.e., a group of related records). The current record needs a unique copy of that DAG in order to complete the DAG associated with the current record.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

- i. extends is a record number for a previously printed record. Any record which contains an extends attribute is a continuation record. The information listed in such a record actually belongs to the record referenced by the extends attribute.

Syntax: positive integer or NULL.

Occurs: at most once.

Default: NULL.

20. undefined Record:

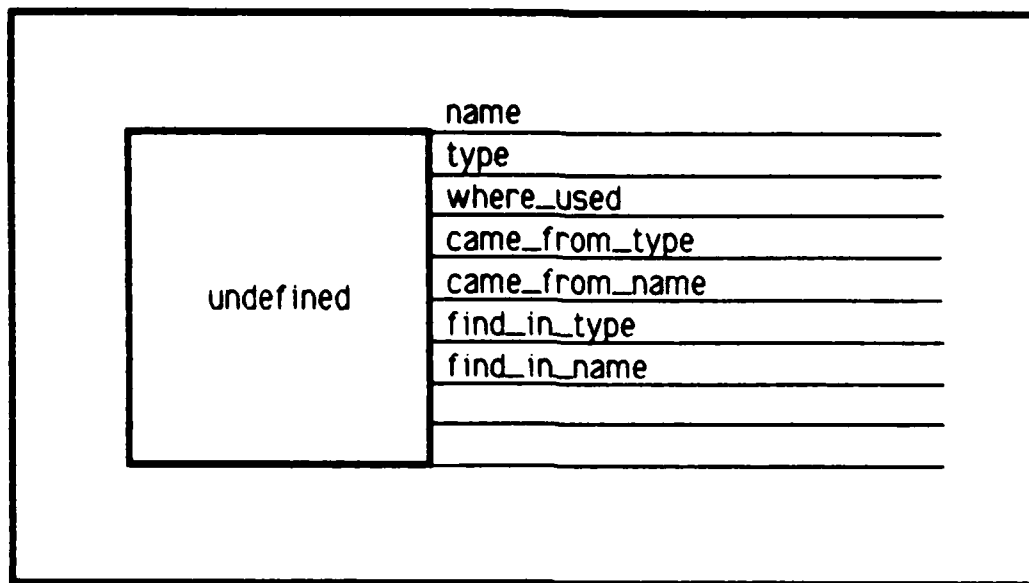


Figure B.22 : Undefined Record.

Record Definition: The undefined record specifies information related to any VHDL identifier which was used in a VHDL source description but was not defined in that description.

Attribute Definitions:

- a. name an undefined VHDL identifier.

Syntax: any valid VHDL identifier or constant.

Occurs: at most once.

Default: NULL.

- b. type specifies how the identifier was used.

Syntax: signal, variable, function, procedure, architecture, package,

configuration, interface, port, or parameter.

Occurs: at most once.

Default: no default.

c. where_used specifies the record number for the VIA record which used the identifier.

Syntax: positive integer

Occurs: at most once.

Default: no default.

d. came_from_type specifies the VHDL construct in which the identifier was used.

Syntax: architecture, package, interface, function, procedure, or configuration.

Occurs: at most once.

Default: no default.

e. came_from_name specifies the name of the VHDL construct which used the undefined identifier.

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

f. find_in_type specifies the VHDL construct in which the identifier was expected to be found.

Syntax: package, interface, configuration, or unknown.

Occurs: at most once.

Default: unknown.

- g. find_in_name the name of the VHDL construct which should define the undefined identifier (if known).

Syntax: any valid VHDL identifier.

Occurs: at most once.

Default: NULL.

21. **viatable Record:**

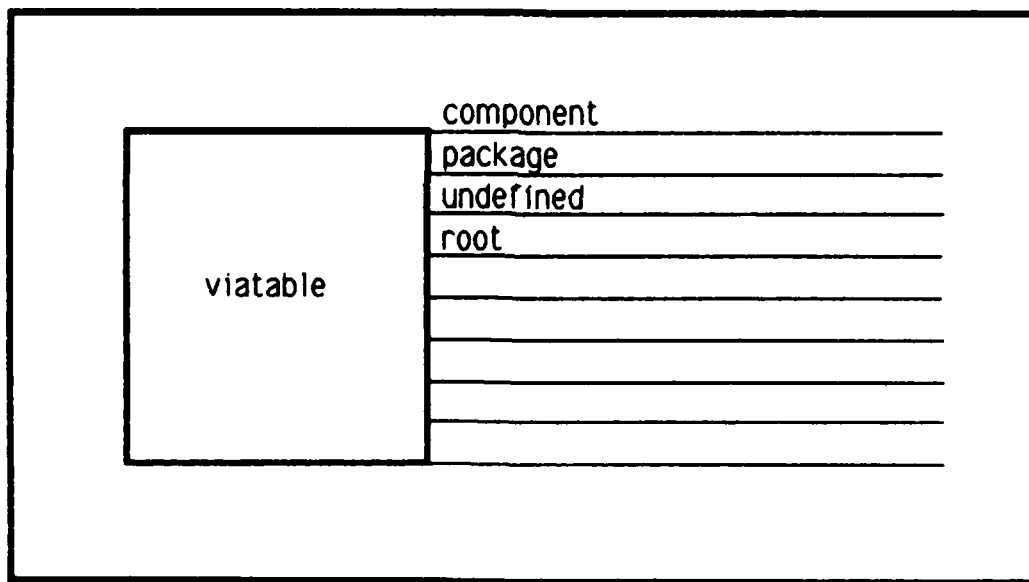


Figure B.23: Viatable Record

Record Definition: The viatable record is the control record for the entire VIA file (see previous discussion on the viatable structure in this appendix.)



Attribute Definitions:

a. component is the record number for a record which specifies a component record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

b. package is the record number for a record which specifies a package record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.



c. undefined is the record number for a record which specifies a undefined record.

Syntax: positive integer or NULL.

Occurs: zero or more times.

Default: NULL.

Appendix C: Example Test Data

This appendix provides a representative set of test cases which were used to verify the language analyzer's function. Each test case followed the general procedures described in Chapter 5. The specific test cases varied with respect to input and output data. Therefore, only VHDL source code, enhanced DDS, and VIA records are depicted for each test case. The VHDL source code shows the language statements which were analyzed. The enhanced DDS depicts the directed acyclic graphs produced from the VHDL statements. The VIA records represent the enhanced DDS. Because of the repetitive nature of the test cases, a discussion of each test case procedure is not provided. The interested reader is referred to the discussion presented in Chapter 5.

Test Case 1.

VHDL Source Code:

```
entity INTERFACE_NAME is  
end ;
```

enhanced DDS:

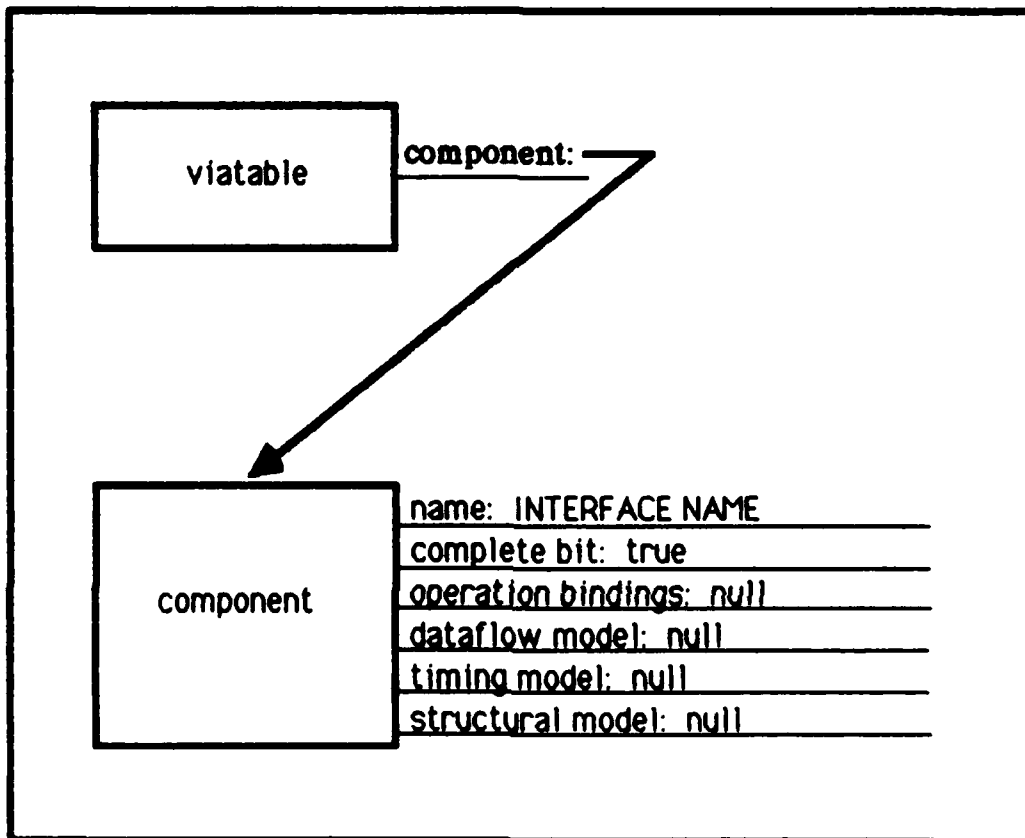


Figure C.1: Test Case 1.

VIA Representation:

```
0 viatable ( component = 1 ; )  
1 component ( name = INTERFACE_NAME ; )
```

Test Case 2.

VHDL Source Code:

```

package PACKAGE_NAME is
  procedure A_PROCEDURE_NAME is
    begin
      null;
    end;
end ;

```

enhanced UDS:

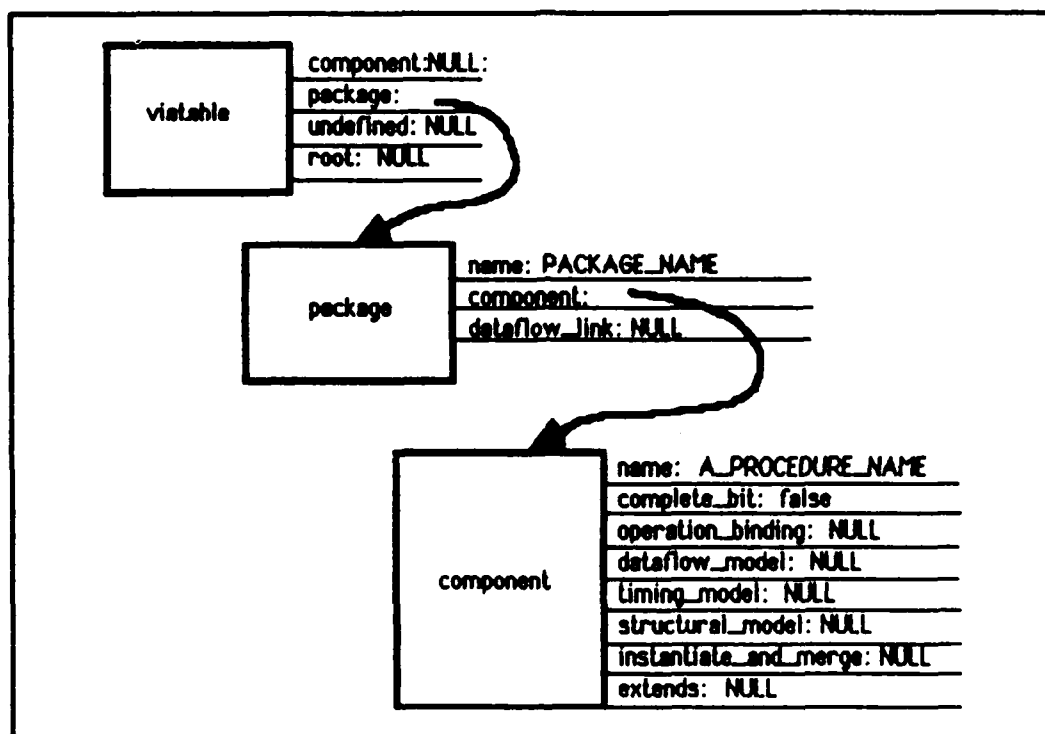


Figure C.2: Test Case 2.

VIA Representation:

```

0 viatable ( package = 1 ; )
1 package ( name = PACKAGE_NAME ; component = 2 ; )
2 component ( name = A_PROCEDURE_NAME ; complete_bit = false ; )

```

Test Case 3.

VHDL Source Code:

```
procedure PROCEDURE_NAME is  
  begin  
    null;  
  end;
```

enhanced DDS:

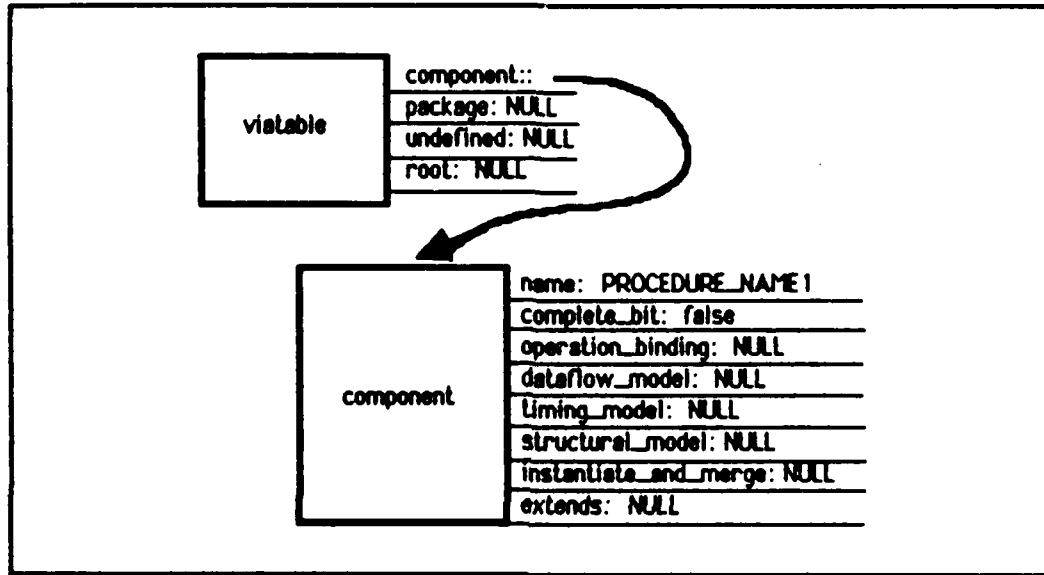


Figure C.3: Test Case 3.

VIA Representation:

```
0 viatable ( component = 1 ; )  
1 component ( name = PROCEDURE_NAME ; complete_bit = false ; )
```

Test Case 4.

VHDL Source Code:

```
function FUNCTION_NAME return A_TYPE_MARK is  
  begin  
    null;  
  end;
```


enhanced DDS:

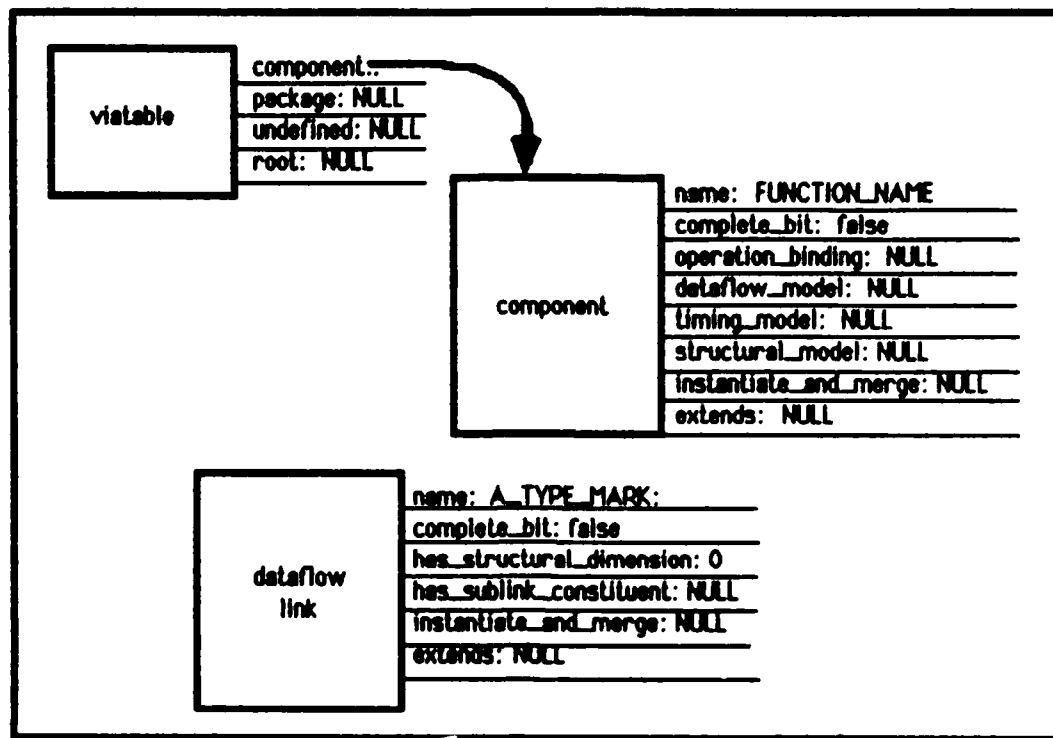


Figure C.4: Test Case 4.

VIA Representation:

```
0 viatable ( component = 1 ; )
1 component ( name = FUNCTION_NAME ; complete_bit = false ; )
2 dataflow_link ( name = A_TYPE_MARK ; complete_bit = false ; )
```

Test Case 5.

VHDL Source Code:

```
architecture ARCHITECTURE_NAME
  of INTERFACE_NAME is
    A_BLOCK_LABEL: block
      begin
        process
          begin
            null;
          end process;
        end block;
      end ;
```

enhanced DDS:

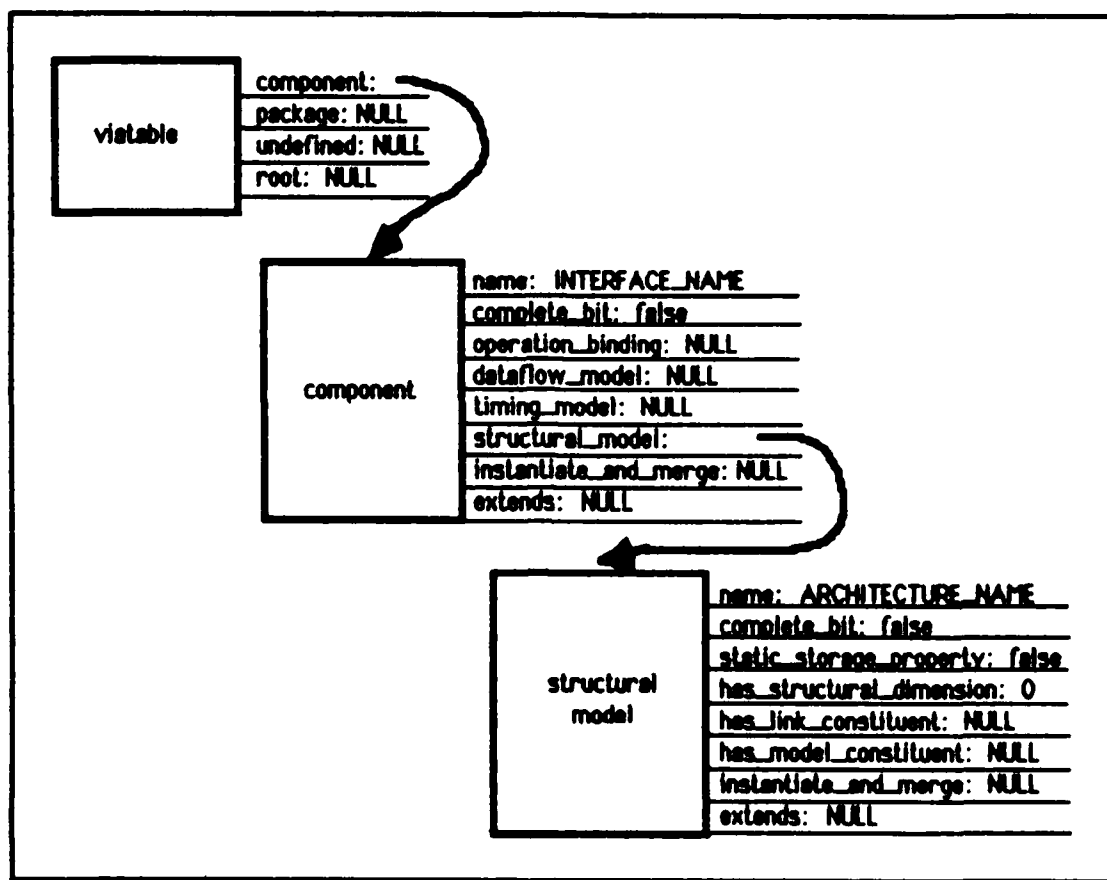


Figure C.5: Test Case 5.

VIA Representation:

```
0 viatable ( component = 1 ; )
1 component ( name = INTERFACE_NAME ; structural_model = 2 ; complete_bit
= false ; )
2 structural_model ( name = ARCHITECTURAL_NAME ; complete_bit = false ; )
```

Test Case 6.

VHDL Source Code:

```
configuration CONFIGURATION_NAME
of INTERFACE_NAME
for ARCHITECTURE_NAME is
end ;
```

enhanced DDS:

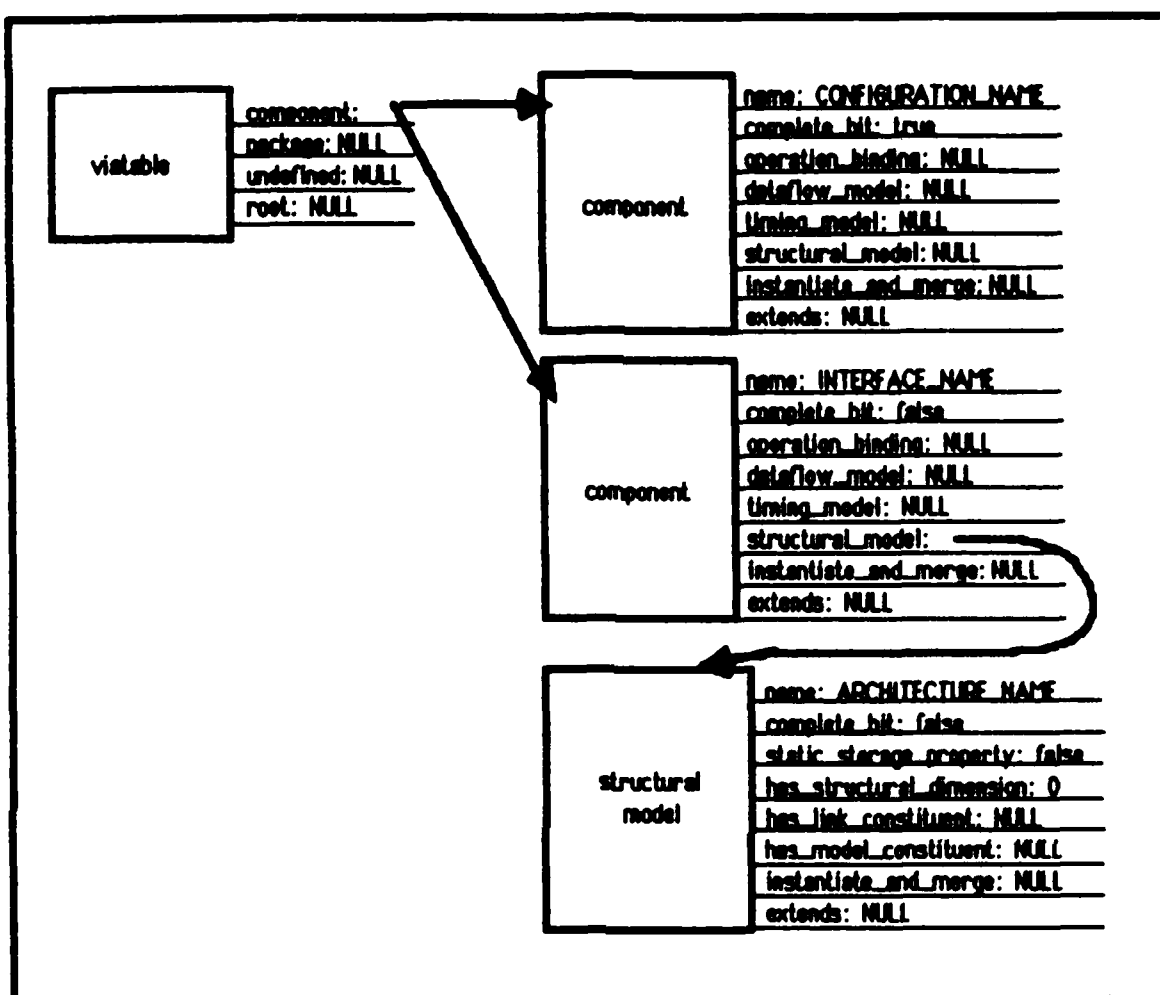


Figure C.6: Test Case 6.

VIA Representation:

```

0 viatable ( component = 1 ; component = 2 ; )
1 component ( name = CONFIGURATION_NAME ; )
2 component ( name = INTERFACE_NAME ; structural_model = 3 ; complete_bit
= false ; )
3 structural_model ( name = ARCHITECTURE_NAME ; complete_bit = false ; )
  
```

Bibliography

Afsarmanesh, Hamideh and others. An Extensible Object-Oriented Approach to Databases for VLSI/CAD, Technical Report CRI-85-09, 7 October 1985. Contract F49620-81-C-0070. Department of Electrical Engineering-Systems, University of Southern California, Los Angeles CA, 23 April 1985.

Air Force Wright Aeronautical Laboratories (AFWAL). VHSIC Hardware Description Language (VHDL) Standardization Effort. Letter attachment. 12 February 86.

Arnold, Michael. "LYRA(CAD)", UNIX Programmer's Manual, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley CA, 20 January 1986.

AT&T Information Systems. The UNIX System User's Manual. Englewood Cliffs: Prentice-Hall, 1986.

Aylor, J. H., R. Waxman, and C. Scarratt. "VHDL -- Feature Description and Analysis," IEEE Design and Test of Computers, 3: 17-27 (April 1986).

CAD Language Systems, Inc., VHDL Language Reference Manual, IEEE Preliminary Version. Rockville: CAD Language Systems, Inc., 28 June 1986.

Carter, Lt Col Harold W. Lecture materials distributed in VHDL meeting. School of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1985.

Dewey, Capt Allen and 1Lt Anthony Gadiant. "VHDL Motivation," IEEE Design and Test of Computers, 3: 12-16 (April 1986).

Drew, Daniel. Lecture materials distributed in Programming Languages. Department of Industrial Engineering, Texas A&M University, College Station TX, 1981.

Fairley, Richard E. Software Engineering Concepts. New York: McGraw-Hill, Inc., 1985.

Fitzpatrick, Dan. "MEXTRA(CAD)", UNIX Programmer's Manual, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley CA, 28 February 1983.

George, Capt Bruce. Discussion. School of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, August 1986.

Intermetrics, Inc. VHDL Analyzer Program Specification. Contract F33615-83-C-1003. Bethesda MD, 30 April 1986.

----. VHDL Language Reference Manual Version 7.2. Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

----. VHDL User's Manual Volume I -- Tutorial. Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

----. VHDL User's Manual Volume II -- User's Reference Guide. Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

Johnson, Stephen C. Yacc: Yet Another Compiler-Compiler. Murray Hill: Bell Laboratories, 31 July 1978.

Katzenelson, Jacob and Eliezer Weitz. "VLSI Simulation and Data Abstractions," IEEE Transactions on Computer-Aided Design, CAD-5: 371-378 (July 1986).

Knapp, David W. and Alice C. Parker. A Data Structure for VLSI Synthesis and Verification. Contract DAAG29-80-k-0083. Department of Electrical Engineering-Systems, University of Southern California, Los Angeles CA, 8 May 1984.

Kulp, P. Private communication. School of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 10 April 1986.

Lesk, M. E. and E. Schmidt. Lex - A Lexical Analyzer Generator. Murray Hill: Bell Laboratories, 31 July 1978.

Linderman, Capt Richard. Discussion. School of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, August 1986.

Ousterhout, John. Editing VLSI Circuits with Caesar. Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley CA, 22 March 1983.

Texas Instruments, Inc., Design Utility Systems. VHDL Design Library Specification. Contract F33615-83-C-1003. Texas Instruments, Inc., Dallas TX, 30 July 1984.

Schreiner, A. V., and H. G. Friedman, Jr. Introduction to Compiler Construction with UNIX. Englewood Cliffs: Prentice-Hall, Inc., 1985.

Vladimirescu, A., and others. SPICE User's Guide. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley CA, 1 October 1983.

Waxman, Ron. "Hardware Design Languages for Computer Design and Test," Computer, 19: 90-97 (April 1986).

VITA

Deborah J. Frauenfelder was born on 2 June 1953 in Manhattan, Kansas. She graduated from West Denver High School in 1971 and enlisted in the Air Force in June 1977. While stationed with the 2851 ABG, Kelly AFB, Texas, she was selected for the Airmen Education and Commissioning Program, and subsequently, transferred to Texas A&M University, College Station, Texas, to complete a Bachelor of Science Degree in Computing Science. After receiving her commission through Officer Training School in August 1982, she was assigned to Headquarters, United States Air Force in Europe, Ramstein AFB, Germany, where she designed software to support intelligence mission requirements. Upon leaving Germany, she was assigned to the Air Force Institute of Technology, School of Engineering at Wright-Patterson AFB, Ohio in May of 1985.

Permanent address: 4740 Dapple Gray Lane
Colorado Springs, CO 80914

A178648

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/MA/86D-1			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENC		7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433			7b. ADDRESS (City, State, and ZIP Code) 1405 E. WOLLAVER Dean for Research and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Air Force Wright Aeronautical Labs		8b. OFFICE SYMBOL (If applicable) AFWAL/AADE		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, Ohio 45433			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) An Implementation of a Language Analyzer for the Very High Speed Integrated Circuit Hardware Description Language					
12. PERSONAL AUTHOR(S) Deborah J. Frauenfelder, B.S.C.S., Capt, USAF					
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1986 December	
				15. PAGE COUNT 172	
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary; include block number)		
FIELD	GROUP	SUB-GROUP			
09	02		VHDL Compiler		
			HDL Hardware Description Language		
			VIA DDS		
19. ABSTRACT (Continue on reverse if necessary; include block number) This thesis describes the incremental approach used to develop the first known C-based, UNIX-supported translator/analyzer for the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). This research consisted of defining a VHDL Intermediate Access (VIA) format as a translation target, dividing VHDL into manageable segments, describing VHDL-to-VIA relationships, designing software modules to create those relationships, and evaluating the functional and performance characteristics of the analyzer. The intermediate form, VIA, was based upon the Design Data Structure (DDS) developed by Alice Parker and David Knapp. Three of the nine VHDL language subsets identified were implemented in the language analyzer. In increments, these subsets were manually translated into specific examples of an enhanced version of DDS represented in a pile file format (VIA). These examples were then used as specifications for designing program modules to automatically translate VHDL code into VIA. After the program modules were written, these same examples were used as formal functional test specifications.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard R. Gross, Lt Col, USAF			22b. TELEPHONE (Include Area Code) (513) 255-3098		22c. OFFICE SYMBOL AFIT/ENC

END

5-87

DTIC