# VHDL2DDS:
# A VHDL Language to DDS Data Structure Translator

Chih-Tung Chen*

CEng Technical Report 91-21

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA. 90089-2562

July 26, 1991

*Supported by the Department of Navy, Contract No. N00039-87-C-0194

## Abstract

This document describes VHDL2DDS, a VHDL (VHSIC Hardware Description Language) to DDS (Design Data Structure) translator. VHDL2DDS is a part of the designer interface of the USC ADAM system, and is used to generate DDS behavioral designs from the parsed VHDL specifications. In order to translate the *von Neumann-type* VHDL description to the *data flow-type* DDS representation, VHDL2DDS uses several analysis techniques known as control flow analysis, data flow analysis, and graph optimization to achieve this process. The objective of the VHDL2DDS is to generate an optimized DDS behavioral design which accurately captures the necessary and sufficient information of the input VHDL specification.

The specific focuses of this document are on the usage of VHDL2DDS, the restricted subset of VHDL, the DDS representation of VHDL constructs, and the interpretation of the VHDL2DDS output. The various analysis and optimization techniques performed by VHDL2DDS are briefly described.



**Key words** - design specification, behavioral design, control flow analysis, data flow analysis, graph optimization.

# Contents

# Chapter 1

# Introduction

The goal of *high-level synthesis* of digital systems is to automate the design process from a behavioral description to the structural implementation, to speed up the design process, and to reduce design errors. The ADAM (Advanced Design AutoMation)[15] system at the University of Southern California is developed to achieve these aims.

The synthesis process typically involves three stages: *description processing*, *structural synthesis* and *physical layout generation*. In the ADAM system, the description processing is done by the Designer Interface subsystem. The design specifications, constraints, and the design library are entered and translated to a unified multilevel design representation called DDS (Design Data Structure)[16] through this subsystem. Then, the Structural Synthesis subsystem and the Physical Layout Generation subsystem (future) can be invoked in sequence to transform the behavioral specifications to RTL (Register-Transfer Level) structures, and later physical layouts.

VHDL2DDS is a part of the Designer Interface subsystem. It is responsible for translating the behavioral specification written in VHDL to the DDS representation.

## 1.1 Overview of VHDL2DDS

In ADAM, the task of VHDL to DDS translation is divided into two parts, *parsing* and *translating*. A commercial VHDL parser[11] from CAD Language Systems Inc. (CLSI) is used to ease the implementation of the VHDL to DDS translation and provide better syntax analysis ability. All VHDL input shall first be parsed and transformed into an intermediate representation[1] by the VHDL parser. VHDL2DDS performs its translation and optimizations on this intermediate structure and finally generates the corresponding DDS representation.

VHDL[14] is much like a programming language. It can be used to express both the behavior and the structure of a hardware design. However, only the behavioral aspect of VHDL is chosen as the input language of the ADAM system. The allowed VHDL constructs are described in Section 2.2. DDS, on the other hand, is more akin to the compiled internal form of a 'design language'. It partitions design information into four *subspaces*. These four subspaces represent respectively the dataflow behavior, the timing behavior, the logical

[1] Basically, it is a *syntax tree*, and is stored in form of the CLSI DLS (Design Library System)[10] information model.

structure and the physical structure of a design. The VHDL to DDS translation performed by VHDL2DDS involves the 'behavioral subspaces' only; namely, the *dataflow* subspace and the *timing* subspace. The objective is to generate a DFG (Data Flow Graph), a CTG (Control Timing Graph) and a set of *bindings* which together represent the same behavior as described in the given VHDL input.

## 1.2 The Problem

Basically, the VHDL to DDS translation is similarly to the translation[3] of a von Neumann-type high-level language to a data flow language[1, 6]. They are both required to analyze the data dependencies and the control flow which exist between statements in a high-level program and to construct a graphical form of code which describes the same behavior as in the high-level program. The main difference is that the order of execution of a data flow program is implied by the partial ordering of the computations and the data availability, whereas in DDS, the order of execution of a DFG is stated in the associated CTG through operation bindings between the DFG and CTG.

The major difficulty faced in this kind of translation is how to map variables of a high-level program into values of a *single-assignment* DFG. In general, a variable may have more than one value associated with it during its lifetime. In order to track the uses of every variable in the program, some data flow analysis technique[2, 5, 13] must be used to identify the data dependencies between statements. This problem is complicated even more when control flow constructs such as conditional statements, loops and branches are involved.

In fact, the VHDL to DDS translation is even harder due to several additional constraints imposed by DDS. First, the DFG must be *acyclic*, which makes loops and branches difficult to be handled because feedback edges are not allowed. Next, all control information must go to CTG; as a result, the control flow constructs in VHDL have to be modeled by several pieces of the DFG with proper bindings to the associated CTG.

## 1.3 The Approach

As discussed in the previous section, there are two major issues to be dealt with; namely, how to analyze data dependencies and how to model control flow constructs in DDS. In addition, these two issues are inter-related, and cannot be solved independently. To make the problem tractable, VHDL2DDS uses the following approach:

1. develop a control flow analysis procedure[2, 4] to extract the control information from a given VHDL description and construct a *flow graph*[2] for later data flow analysis. A flow graph is a directed graph. Each node, called a *basic block*, in the graph represents a sequence of computations, and each edge represents the flow of control. The flow of control enters at the beginning of a basic block and leaves at the end without halt or possibility of branching in the middle. The reasons for building the control flow graph are to separate the computations from the control flow and to enable efficient analysis and code generation techniques to be applied.

2. Since the flow graph isolates the computations within a basic block from the control flow information, the data flow analysis can be performed in two steps. First, we use a local data flow analysis procedure to collect intra-block data dependencies. Then, a global data flow analysis procedure is used to analyze the inter-block dependencies. After this phase adding data flow information, the annotated flow graph becomes a combination of a DFG and a CTG.

3. Though the annotated flow graph carries enough information to generate the DFG and the CTG, it may contain a lot of copy operations of the form $a = b$ which must be eliminated in order to produce an optimized[2] DFG. A technique called *value tracing* and a set of graph reduction rules are developed for this need.

4. Having set up the annotated flow graph, the code generation becomes straightforward. It is done in two passes. In the first pass, the CTG is produced, and information about DFG and bindings is collected as well. Finally, the DFG and operation bindings are generated during the second pass.

The above discussion is only an overview to the VHDL to DDS translation. It's too brief to fully understand how the translation works. However, it's the basic knowledge which must be kept in mind by the users of VHDL2DDS. For more on this topic, refer to [8].

[2]It does not mean optimal because only *copy propagation elimination* is performed by VHDL2DDS. However, the flow graph can facilitate all other optimizations[2] such as *constant folding*, *code motion*, *code hoisting*, etc.

# Chapter 2

# Using VHDL2DDS

This chapter is devoted to the use of VHDL2DDS. The emphases here are on the requirements of the VHDL to DDS translation, the VHDL input and its restrictions, and the DDS representation of various VHDL constructs. No attempt is made to describe the VHDL language, the DDS data structure, and how to invoke VHDL2DDS or the VHDL parser. For details on this topics, see the proper documents[14, 16, 9, 11].

## 2.1 Requirements

A successful VHDL to DDS translation depends on many prerequisites. In fact, some of the requirements are not imposed by VHDL2DDS but by the specification of the USC ADAM system.

- Only the *behavioral type* VHDL description is allowed, and its syntax must conform to the restrictions described in Section 2.2.
- All VHDL descriptions have to be parsed by the CLSI VHDL parser prior to running VHDL2DDS. The parsing task is expected to be integrated to VHDL2DDS in the future.
- After parsing a VHDL description, an intermediate data is generated for each *design unit* within the description. Each design unit has to be translated by VHDL2DDS separately. The order in which these design units are translated must be consistent with the partial ordering defined by the following rules:

  1. A *primary unit* whose name is referenced within a given design unit must be translated prior to the given design unit.
  2. A primary unit must be translated prior to the translation of its corresponding *secondary unit*.

  where the primary unit is either an entity declaration or a package declaration and the secondary unit is a separately analyzed body of a primary unit such as an architecture body or a package body.

- The output of VHDL2DDS shall be stored in a DDS database. No other output format is supported by the current version of VHDL2DDS.
- In general, the DDS graphs generated by VHDL2DDS are hierarchical. They must be further processed by the *flattener* and *colorer* programs before being used by the *synthesis* and *area estimation* subsystems.

As one might expect, these requirements introduce unnecessary overhead to the translating task. Some future enhancements to VHDL2DDS are discussed in Chapter 4 to improve the usability of the ADAM VHDL interface.

## 2.2 The VHDL Subset

The VHDL used in ADAM is a subset of the IEEE Standard VHDL[14] since we are only concerned with representing behavioral specifications in VHDL. This subset was carefully defined to devoid features incompatible with the notion of behavioral description or unable to be represented in DDS, while still giving sufficient expressive power for most applications.

The allowed VHDL constructs are limited to the following:

1. *Design Entities*

   The primary hardware abstraction in VHDL is the *design entity*. A design entity is defined by an *entity declaration* together with a corresponding *architecture body*.

   - *Entity Declarations*
     The entity declaration basically defines the inputs and outputs of the design entity. A given entity declaration is restricted to be used by only one design entity; that is, it can not be shared in this VHDL subset. The restrictions described in Item 5 are applied accordingly to the *entity header* and the *entity declarative part* of a given entity declaration[1]. The *entity statement part* must be empty in each design entity; in other words, the behavior of a design entity must only be specified in the corresponding architecture body.
   - *Architecture Bodies*
     There are three general styles of descriptions possible within an architecture body: *structural*, *dataflow* and *behavioral*. However, only the behavioral one is allowed. Behavioral descriptions specify data transforms in terms of algorithms for computing output responses to input changes. The feature of multiple asynchronous processes is not yet supported in the current version of VHDL2DDS; therefore, each architecture body is required to have one and only one concurrent statement in the *architecture statement part*.

2. *Subprograms*

[1] In fact, this rule is applied to all declarative parts in this VHDL subset. It will not be stated explicitly in the rest of the VHDL subset definition unless additional restrictions are required.

Since the *configuration* is not included in the VHDL subset, *subprograms* shall serve as the major mechanism for building the desired design hierarchy[2]. The definition of a subprogram can be given in two parts: a *subprogram declaration* and a *subprogram body*. Subprograms without subprogram bodies shall be used as the break-away points for the design hierarchy or the interfaces of the modules in the system library. Both *procedures* and *functions* are allowed. The *subprogram overloading* is not supported, and the *operator overloading* is limited to once for each scope of declarations.

3. *Packages*

   Packages provide a means of defining declarations which can be shared by different design units. One of the major usages of packages in ADAM will be to define the interfaces of some implementation-dependent module libraries. In such a case, the *package declaration* has no corresponding *package body*. No special restrictions except those in Item 5 are imposed on packages.

4. *Types*

   In VHDL, a type is characterized by a set of values and a set of operations. All implicitly declared operations for a given type declaration are supported and will be translated automatically by VHDL2DDS. However, they are not recommended to be used in the VHDL descriptions because there may be no corresponding modules in libraries for bindings. As a result, the explicitly declared subprograms for a type are more appropriate in terms of module bindings. Two classes of types are allowed with restrictions; namely, *scalar* types and *composite* types.

   - *Scalar Types*
     Scalar types are limited to the predefined types BIT, BOOLEAN, and INTEGER only. Currently, users can not define their own scalar types. The INTEGER type is assumed to be a 32-bit implementation.
   - *Composite Types*
     The composite type is the only user-definable type class in this VHDL subset. It is further limited to *array* types only. An array object is a composite object consisting of elements that have the same type. Its primary usages are to model different bit-width values and memories. The maximal dimensionality of an array type is limited to 2. Both *unconstrained array* types and *constrained array* types are allowed. The *index definition* of an unconstrained array type must be INTEGER, and the *index constraint* of a constrained array type must be *ranges*. BIT_VECTOR is the predefined array type supported by VHDL2DDS.

     Since *subtypes* are not supported, there are several limitations on the uses of array types. First, it is not allowed to define a constrained array type from an existed unconstrained array type. This makes the unconstrained array types of little use. A constrained array type is no longer defined as an unconstrained array type and a subtype of this type. It itself is a 'type'.

[2]In fact, this limitation makes the design entity unsuitable for describing *internal* blocks because there is no way to bind a collection of design entities into a design hierarchy without using a *configuration declaration*.

For each array type, two additional operations are implicitly defined by this VHDL subset. They are *array read* operations and *array write* operations. If an *indexed name* appear at the right (left) hand side of an assignment statement, an array read (write) operation will be used. This feature is well suited for modeling memories; however, it can not model the extraction of a subvalue from a multi-bit value.

5. *Declarations*

   In addition to design entities, subprograms, packages and types, the other kinds of declarations allowed are *object* declarations and *interface* declarations.

   - *Object Declarations*
     All three classes of objects are allowed; namely, *constants*, *signals*, and *variables*. An object declaration declares an object of a specified type. The feature of *deferred constants* is not supported. Signals will be treated as variables; that is, only the syntactical aspect of signals is preserved, but their semantics will be identical to variables in terms of the VHDL to DDS translation. Therefore, a signal declaration is not allowed to have a *resolution function*, *guards*, and the *signal kind*.
   - *Interface Declarations*
     Interfaces objects include constants, signals, and variables, too. The restrictions described above are applied accordingly. In addition, the *mode* of an interface object is limited to either **in** or **out**.

6. *Names*

   All forms of names, except *attribute* names and *slice* names, are allowed. The identifier for an entity, a package, a subprogram, or an interface object shall have only first 5 characters significant after translation. An index name is considered to be an array read (write) operation instead of simply denoting an element of an array.

7. *Expressions*

   An expression is a formula that defines the computation of a value. It consists of a set of operators and their operands. Though all VHDL predefined operators are supported by VHDL2DDS, VHDL2DDS does not assume any specific implementation to a predefined operator, nor is it aware of the availability of any library module for binding. Care must be taken not to use any predefined operator unless the user can make sure there exists some corresponding module in the library or the operator in question will somehow be implemented. In ADAM, a more appropriate approach will be to define a package for each available module library using function declarations or overloaded operators and use these functions or operators in expressions instead of predefined ones.

   The allowed operands in an expression include names, literals, and function calls. In addition, an expression enclosed in parentheses may be an operand in an expression. A literal is either a integer literal, a boolean literal, a bit literal, or a bit string literal.

8. *Sequential Statements*

   Sequential statements shall be the major means for describing the behavior of the component under design. The allowed sequential statements are:

   - *Signal assignment* statement.
   - *Variable assignment* statement.
   - *Procedure call* statement.
   - *If* statement.
   - *Case* statement.
   - *Loop* statement.
   - *Next* statement.
   - *Exit* statement.
   - *Return* statement.
   - *Null* statement.

   Statement labels can be used whenever necessary. A signal assignment statement is considered like a variable assignment statement. Hence, *transport* delay is not supported and the waveform at the right hand side can consist of one element only. In addition, a waveform element is not allowed to have a **after** clause. The *iteration scheme* of a **for** loop must be a *range* of type INTEGER.

9. *Concurrent Statements*

   The *process* statement is the only form of concurrent statement allowed in this VHDL subset[3]. A process statement defines an independent sequential process representing the behavior of the design. The execution of a process statement is modeled by the endlessly repetitive execution (an implicit loop) of its sequence of statements. Hence, a process statement is not allowed to have a *sensitivity list*.

## 2.3 Representing VHDL in DDS

In this section, the correspondence between VHDL and DDS is given. Understanding this section relies on a basic knowledge of DDS. The major difficulty in representing a VHDL description in DDS is they (VHDL and DDS) use 'incompatible' models for representing the behavior of a design. Basically, there is no one-to-one relationship between them. Though, by using extensive flow analysis techniques, VHDL2DDS can generate a DDS representation which mimics the behavior of a given VHDL design, it is not easy to know what the output of VHDL2DDS will be. Therefore, the objective here is to give the VHDL designers some extent of ability to predict the output of VHDL2DDS.

[3]Since the feature of multiple concurrent processes is not supported, the inclusion of the process statement in this VHDL subset is merely for syntactical reasons.

Before the discussion of the relationship between VHDL and DDS, we assume the readers already have some basic knowledge of DDS[16]. The discussion will be organized according to various VHDL constructs, and the DDS terminology shall be used without further explanation.

### 2.3.1 Declarations

Most of the VHDL declarations have obvious counterparts in DDS. In general, they are translated into the definition part of a DDS representation; e.g., the *value definitions* and the *operation definitions* of the DDS dataflow subspace[4]. The generalized scheme is enumerated below:

1. *Design entities*

   The fundamental structure of DDS is the *component*. A component under design is identified by an entity declaration and its associated architecture body. For each entity declaration being translated, a DDS component is created, and the interface objects declared in the entity header become the input (output) *values* of the component's DFG. The translation of an entity declaration defines the primary inputs and outputs of the component only. The DFG body and the CTG are null until the associated architecture body is translated. The process statement within the architecture body provides the behavioral information of the design to fill the DFG and CTG.

2. *Subprograms*

   Subprograms are also transformed into DDS components. In fact, subprograms and design entities are treated in the same way by VHDL2DDS, where subprogram declarations correspond to entity declarations and subprogram bodies to architecture bodies. However, the component being generated from a subprogram usually is not the final component under design. The primary use is for building the design hierarchy and defining the interfaces of some implementation-dependent operations (modules).

3. *Packages*

   A package is a *declarative region* in which a set of sharable declarations are grouped. It has no physical meaning, and is primarily related to the *scope* and the *visibility* of declarations. On the other hand, a single uniform *naming space* is shared by all objects created in DDS. That is, every object in DDS is globally 'visible'. In order to represent the concept of packages and the visibility of names in DDS, the uniform naming space is divided into several naming *subspaces*. A package, a design entity, or a subprogram defines a naming subspace in DDS. In general, declarations within different declarative region in VHDL shall reside in different naming subspaces in DDS[5].

[4]Values and operations are sometimes termed *links* and *nodes* respectively in DDS.

[5]For example, an identifier named 'X' defined in both package 'A' and package 'B' could be named differently as 'A.X' and 'B.X' in DDS. For details on the naming convention used by VHDL2DDS, refer to Appendix A.

4. *Types*

   A type in VHDL corresponds to a dataflow *value definition* in DDS. Both of them are used to define a set of possible values for a particular application. The value definitions provide the characteristic information, e.g., the bit width, to the values in a DFG, which is essential to several synthesis steps such as register allocation and memory allocation.

   Each predefined scalar type is translated into a primitive value definition according to its characteristics. For example, BOOLEAN is assumed to be the type of 1-bit flag and INTEGER uses a 32-bit implementation. An array type is transformed into a hierarchically-defined value definition which is organized like a tree structure. The elements of an array type are represented by the *constituents*[6] of the corresponding value definition[7]. The **kind** of the constituents is defined by another value definition derived from the array element type. For a constrained array type, the *structural dimension* is calculated by multiplying the first index range by the structural dimension of the element type.

5. *Objects*

   The DFG in DDS is a *single assignment* graph; that is, a value can only be generated once. This is in contrast to the common notion of a variable that may have more than one value associated with it during its lifetime. Instead, an object is modeled by a sequence of values, each of which is bound to a different range of the associated CTG. The determination of the values for an object depends on how the given object is used in the statement part. Each new assignment to an object defines a value in the DFG. In general, this information is collected by the dataflow analysis. Except the interface objects, the declaration of an object does not mean there will be a value in the DFG. It simply provides the value definition (type) information to those values that will be bound to the object.

### 2.3.2 Statements

In VHDL, the behavior of a design is described algorithmically by a sequence of statements. These statements are contained in a process statement or a subprogram body. As discussed in Section 1.3, to translate a sequence of statements into a equivalent DDS representation several analysis techniques must be applied. First, the sequence of statements are partitioned into several *basic blocks* in order to build the flow graph. Each basic block represents a sequence of consecutive statements without the possibility of branching in the middle. The basic blocks are identified as following:

- Determine the set of *leaders* in the sequence of statements.

[6]They are sometimes called *sublink constituents* or *subvalues*. Basically, the constituents of a value definition are a set of *value references* defined by some other value definitions.

[7]Due to the limitation of the current DDS schema, the constituents are only created for the bounding elements of a constrained array type. For an unconstrained array type, an abstract constituent is used to represent all possible elements.

| *Type* | *Input and Output Sets for Statement S* |
|---|---|
| assignment | $I(S) = \{x \mid x$ is referenced by $S\}$<br>$O(S) = \{x \mid x$ is defined by $S\}$ |
| procedure call | $I(S) = \{x \mid x$ is an input parameter of the procedure$\}$<br>$O(S) = \{x \mid x$ is an output parameter of the procedure$\}$ |
| condition | $I(S) = \{x \mid x$ is referenced by $S\}$<br>$O(S) =$ null |

Table 2.1: Calculation of input and output sets for a statement.

  - The first statement of a basic block is a leader.
  - Any statement that is the target of a conditional or unconditional branch is a leader.
  - Any statement that immediately follows a branching statement is a leader.

- For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the given statement sequence.

By constructing the flow graph, the computations are confined to the basic blocks and the control flow information is represented by the inter-block relationship[8].

1. *Basic Blocks*

   A basic block is a computational unit. The output of a basic block is deterministic in terms of its inputs. The only forms of statements which exist in a basic block are assignment statements, procedure calls, and conditional expressions. Each statement can be modeled as a set of input values, a set of operations, and a set of output values. The calculation of the input and output sets for a statement is illustrated in Table 2.1. The input and output sets for basic blocks are somewhat more complicated. Let $B = S_1, \cdots, S_n$ be any basic block. Its input and output sets are defined to be

$$I(B) = I(S_1) \bigcup \left\{ \bigcup_{i=2}^{n} \left( I(S_i) - \bigcup_{j=1}^{i-1} O(S_j) \right) \right\}$$

   and

$$O(B) = \bigcup_{i=1}^{n} O(S_i)$$

[8]In other words, the edges of the flow graph.

This means that the input set for $B$ consists of values which are used before their redefinition and the output set contain all values defined within the block.

The definition of an input value can be found by a backward scan of the preceding statements until the value appears either in an output set of a statement or in the input set of the enclosing basic block. For each output value, a list of statements where that value is used before its redefinition can be found similarly by a forward scan. Following the analysis described above, it is conceptually easy to generate the DFG for a basic block. The inter-statement dependencies have been established by the use and definition analysis, and the intra-statement dependencies are established according to the syntax trees of the expressions within the statement. By transforming each operation[9] into a *node* and each data dependency into a *link* between nodes, the DFG is built.

Basically, the CTG generated by VHDL2DDS is an initial scheduling of operations and values in the corresponding DFG. The underlying model has been simplified and assumes no time and cost constraints. The objective here is to produce a correct CTG which provides all necessary control and timing information. At later stages of synthesis, the DFG can be rescheduled or the CTG can be improved according to the given constraints.

In this model, each statement corresponds to one or more sequential *ranges*[10] in the CTG. The ranges of a statement are generated as following:

- The expressions within a statement are evaluated from left to right. Each of them is scheduled separately and sequentially. For each expression, it is evaluated by traversing its syntax tree in postorder.
- According the evaluation order of an expression, all function calls are bound to their own ranges, and the primitive operations[11] up to the first function call or between any two function calls are bound to the same range.
- All ranges are concatenated sequentially according to the evaluation order.

Each value is considered to be live from the range where the operation, which defines it, is bound to the range where the last operation, which uses it, is bound. The final CTG of a basic block is built by concatenating sequentially the ranges of all statements within the block.

2. *Conditional Statements*

   Conditional statements include **if** statements and **case** statements. The case statement is transformed into an if statement and no further reference to it is made in the following discussion. If any **elsif** part is present in an if statement, the if statement is viewed as a nested *if-then-else* construct. By doing these transformations, the translation of a conditional statement is reduced to the analysis of simple if-then-else constructs.

[9] It can be a call to an operator, a function, or a procedure.

[10] A range can be viewed as a *time step* in terms of scheduling.

[11] A primitive operation is a call to an operator or a function without a function body.

Basically, an if-then-else construct consists of a *condition*, a *then body*, and an optional *else body*. The condition is a Boolean expression which determines the flow of control. Both the then body and the else body are sequences of statements[12].

To represent an if-then-else construct in DDS, we collect the dataflow and control-timing information from the condition, the then body, and the else body respectively, and build the DFG and CTG by gluing[13] these subgraphs together as show in Fig. 2.1.

Two special operations, *distribute* and *join*, have been introduced into the DFG. These operations allow us to describe the conditional sequencing in the DFG; that is, they are the 'control' operations in the data flow subspace. Their presences are required by the *colorer* program to detect mutually exclusive operations and values in a DFG. An analogy of these operations can be drawn to the *switch* and *select* operations in some data flow languages[12]. However, they do not necessarily correspond to a piece of hardware after the synthesis. A distribute operation is created for each condition-dependent value[14] used by the if-then-else construct, and a join operation is used for each value that is defined by both the then body and the else body.

The CTG of an if-then-else construct is straightforward. A pair of *or-fork* and *or-join* points are created to form a 2-way branch. The choice of branch is based on the value of the condition that is attached to the ranges emanating from the or-fork point. The True branch and the False branch are actually the sub CTGs representing the then body and the else body. In fact, the CTG provides another way, perhaps easier, to detect the mutual exclusive operations and values in the DFG by using the operation bindings. For example, if two operations are bound to the True branch and the False branch respectively, they are certainly mutual exclusive.

3. *Iterative Statements*

   In VHDL, an iterative statement consists of a *loop body* and an optional *iteration scheme* (**while** or **for**). To simplify the analysis, the for loop is transformed into a while loop, and a loop without an iteration scheme is considered as a while loop with a True value as the condition. Therefore, every iterative statement is modeled by a *while* construct. The loop body of a while construct is a sequence of statements that is to be executed repeatedly, zero or more times, until the condition becomes false.

   The DDS representation describes the while construct using an $\alpha$-$\omega$ loop in the CTG with a distribute operation and a join operation in the DFG. Fig. 2.2 shows the generalized graph template of a while construct. Both the CTG and the DFG of a while construct can be viewed to consist of three parts: loop entrance, loop exit, and the loop body. In DFG, the loop entrance is for the evaluation of the loop condition, and the loop exit is a set of distribute operations for loop-dependent values[15]. On the other

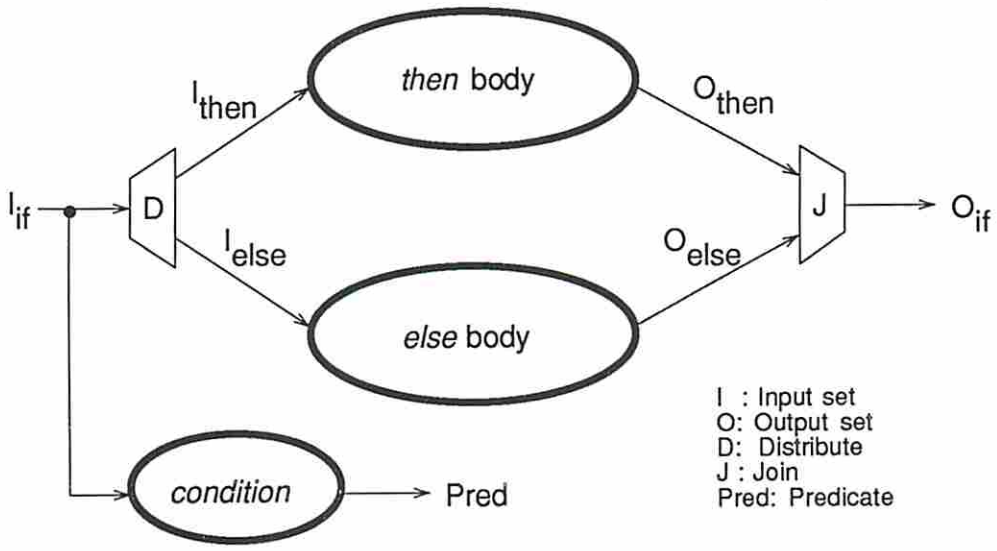
[12]Readers should recall the translation of a sequence of statements discussed in the beginning of this section.

[13]The actual process is not as simple as it sounds. It depends on the global flow analysis which has been intentionally left out to simplify the discussion.

[14]Currently, a condition-dependent value is assumed by VHDL2DDS to be the value used by both the then body and the else body.

[15]A loop-dependent value is considered to be the value used and redefined by the loop body.

DFG:




CTG:

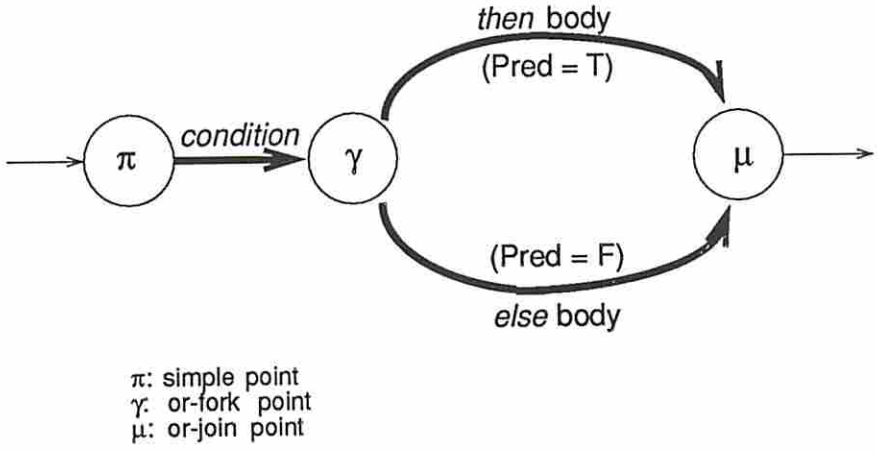



Figure 2.1: The DDS template of an *if-then-else* construct.

hand, the $\alpha$ point in the CTG represents the loop entrance, and the or-fork point in the middle is the loop exit which branches to either the $\omega$ point or whatever follows the loop.

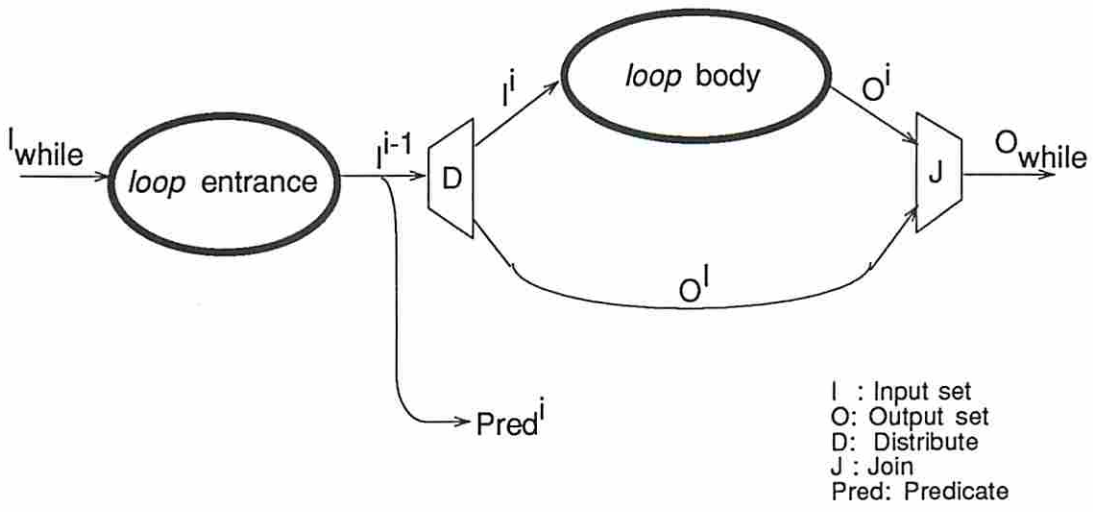
If next or exit statements are used within the loop body, they are considered as either conditional or unconditional branches, and are handled by the basic block analysis. In another view, a conditional next or exit statement is transformed into an if-then-else construct where the else body is the remaining statements in the loop body. The then body is null in an exit statement or a branch to the loop entrance in a next statement. Next or exit statements can easily result in intertwined conditional paths in the DFG and CTG if they are not used in a structured way. The resulted graph cannot be colored properly to detect mutual exclusion. This problem can be solved by using proper loop transformation techniques[18]. However, no attempt is made by the current VHDL2DDS to do loop transformations of any kind.

Looping traditionally involves the re-use of values and operations, but this cannot be done in a single-assignment DFG. The way we indicate the correspondence between the successive operations and values of a loop is by the use of *subscripts*. A subscript indicates a temporal sequence of values or operations; e.g., a stream of values could be described as $b_0, \cdots, b_N$ where $b_i$ would precede $b_{i+1}$ in a sequence. An analogy of subscripts would be the *tags* proposed in[7]. A *fixed loop*[16] can be easily handled by assigning constant subscripts to values and operations in DFG. In fact, this loop can be viewed to be *unrolled* in the DDS. Unfixed loops, on the other hand, cannot be unrolled. In order to deal with this situation, a *symbolic* subscript[17] is used. Any value or operation may have attached a subscripting symbol or expression, such as $x_i$ and $x_{i-3}$, in order to be made explicitly distinguishable.

In the current VHDL2DDS, all loops are assumed to be unfixed loops. A symbolic subscript is defined as $i : 0 \cdots I$, where $I$ denotes the final value of the subscript and $i$ represents the current iteration. For example, if a variable $b$ has an initial value before the loop begins and is redefined within the loop body, then $b_0$, $b_{i-1}$, $b_i$, and $b_I$ are used to denote the initial value, the previous iteration value, the current iteration value, and the final value of variable $b$. If loops are nested, the inner loop is given a new subscript plus the one currently in the outer loop.

[16]A fixed loop is a loop with a fixed number of iterations. On the other hand, an unfixed loop is a loop with an unknown number of repetitions.

DFG:




CTG:

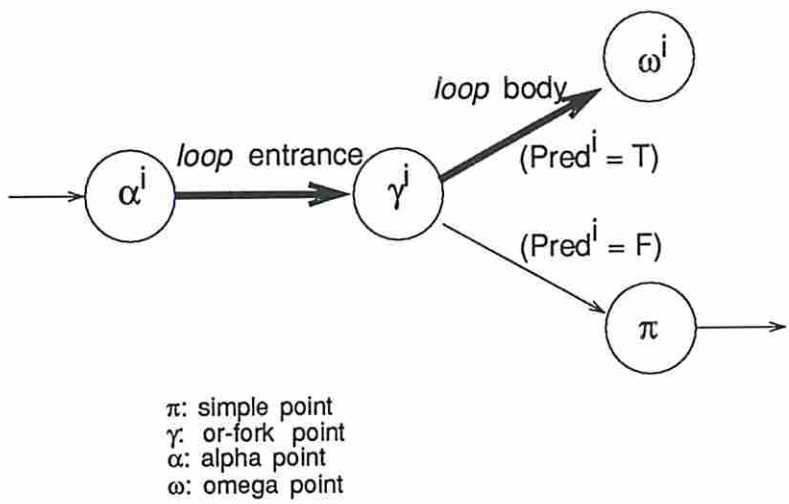



Figure 2.2: The DDS template of a *while* construct.

# Chapter 3

# Example

In this chapter, an example is given to illustrate the uses of VHDL2DDS and its capabilities. This example is a digital filter which serves as a running example in ADAM.

## 3.1 AR Lattice Filter

A major field of the application of the data path synthesis is digital signal processing, where the computations are intensive but regular. Here we use the AR lattice filter as an example.

The AR lattice filter is very regular and suited for supercomputing using systolic array processors. An array element of the AR lattice filter is show in Fig. 3.1. The operations of the four modules in an array element are summarized as the following:

| *Module* | *Operations* |
|---|---|
| A | $a_1 = in_1 * p_1$<br>$a_2 = in_1 * p_2$ |
| B | $b_1 = in_2 + a_2$ |
| C | $out_1 = b_1 * p_3$<br>$c_1 = out_1 * p_4$ |
| D | $out_2 = a_1 + c_1$ |

where $p_1, \cdots, p_4$ are constant parameters and all values and operations are complex.

Before writing a VHDL description for this example, the following assumptions must hold:

- A complex value is represented by a real part and an imaginary part, and both of which are 32-bit integers.
- The standard integer '+' and '*' operators are used and there exist appropriate modules in the library for bindings.
- The constant parameters are supplied externally by the system.

Fig. 3.2 shows a VHDL description for an array element of the AR lattice filter. In this description, there is only one basic block in the architecture body; therefore, only the local

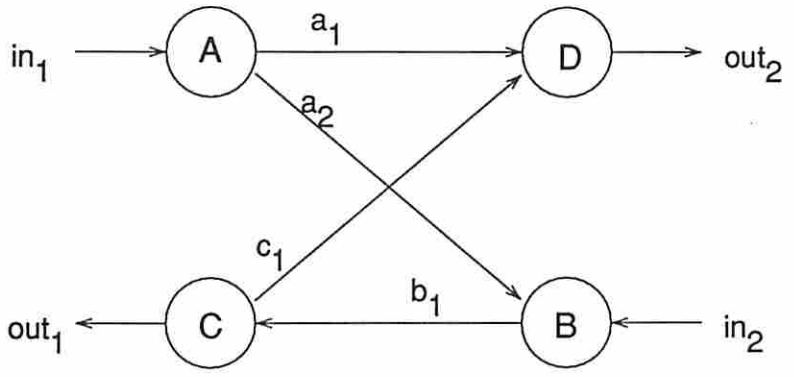



Figure 3.1: An array element of the AR lattice filter.

flow analysis is needed. After collecting the input and output sets of each statement and generating the use and definition sets, we have the use/definition table shown in Table 3.1 Both 'cmult' and 'cadd' procedures are analyzed similarly. From the information provided by the flow analysis, it is easy to generate the DFGs since the use/definition table provides all the inter-statement data dependencies and can be viewed as a linear representation of the DFG. The actual DFGs generated by VHDL2DDS are shown in Fig. 3.3. Some remarks on this figure are worth making:

- The copy statements of the form $x = y$ are eliminated by VHDL2DDS during the graph generation, and the DFGs have been optimized.
- The DFG of the AR filter array element is hierarchical and needed to be flattened.
- Each link in the DFG of the AR filter array element represents a complex value; i.e., it actually consists of two integer values, a real part and an imaginary part.
- The value names and the operation names are simplified and do not reflect the actual identifiers generated by VHDL2DDS. For the exact names, refer to Appendix A.

Since there are no conditional or loop statements in this example, the CTG generated by VHDL2DDS is simply a sequence of ranges, where each range corresponds to a statement in the architecture body[1] and is hierarchically defined by the CTG of either 'cmult' or 'cadd'. The binding table[2] is shown in Table 3.2. In other words, the initial schedule provided by VHDL2DDS is based on the order of the statements in the VHDL description and assumes neither time nor cost constraints.

[1]There are some dummy ranges being generated by VHDL2DDS in the actual CTG.
[2]In this table, the range numbers do not reflect the actual identifiers generated by VHDL2DDS.
[3]This field shows the range in which the value is last used.

```
-- An array element of the AR Lattice Filter using type I complex mutiplier.
entity ARF_ELEM is
 port(
        in1R, in1I, in2R, in2I: in integer;              -- the inputs from neigboring elements.
        out1R, out1I, out2R, out2I: out integer;         -- the outputs to neigboring elements.
        p1R, p1I, p2R, p2I, p3R, p3I, p4R, p4I: in integer      -- the constant parameters.
      );
end ARF_ELEM;

architecture BEHAVIOR of ARF_ELEM is
 -- Type I complex multiplier
 procedure cmult(a, b, c, d: in integer; x, y: out integer) is
 begin
   x := a*c - b*d;                -- the real part
   y := a*d + b*c;                -- the imaginary part
 end cmult;

 -- complex adder
 procedure cadd(a, b, c, d: in integer; x, y: out integer) is
 begin
   x := a + c;                    -- the real part
   y := b + d;                    -- the imaginary part
 end cadd;

begin
 process
   variable a1R, a1I, a2R, a2I: integer;
   variable b1R, b1I: integer;
   variable c1R, c1I: integer;
   variable tmpR, tmpI: integer;
 begin
   -- module A
   cmult(in1R, in1I, p1R, p1I, a1R, a1I);          -- a1   = in1 * p1
   cmult(in1R, in1I, p2R, p2I, a2R, a2I);          -- a2   = in1 * p2
   -- module B
   cadd(in2R, in2I, a2R, a2I, b1R, b1I);           -- b1   = in2 + a2
   -- module C
   cmult(b1R, b1I, p3R, p3I, tmpR, tmpI);          -- tmp  = b1  * p3
   cmult(tmpR, tmpI, p4R, p4I, c1R, c1I);          -- c1   = tmp * p4
   out1R <= tmpR; out1I <= tmpI;                   -- out1 = tmp
   -- module D
   cadd(a1R, a1I, c1R, c1I, tmpR, tmpI);           -- tmp  = a1  + c1
   out2R <= tmpR; out2I <= tmpI;                   -- out2 = tmp
 end process;
end BEHAVIOR;
```

Figure 3.2: A VHDL description for an array element of the AR lattice filter.

| *Statement* | *Input Set* | | *Output Set* | |
|---|---|---|---|---|
| | val | def | val | use |
| 1: $a_1 = in_1 * p_1$ | $in_1$ | 0 | $a_1$ | 7 |
| | $p_1$ | 0 | | |
| 2: $a_2 = in_1 * p_2$ | $in_1$ | 0 | $a_2$ | 3 |
| | $p_2$ | 0 | | |
| 3: $b_1 = in_2 + a_2$ | $in_2$ | 0 | $b_1$ | 4 |
| | $a_2$ | 2 | | |
| 4: $tmp = b_1 * p_3$ | $b_1$ | 3 | $tmp$ | 5, 6 |
| | $p_3$ | 0 | | |
| 5: $c_1 = tmp * p_4$ | $tmp$ | 4 | $c_1$ | 7 |
| | $p_4$ | 0 | | |
| 6: $out_1 = tmp$ | $tmp$ | 4 | $out_1$ | 0 |
| 7: $tmp = a_1 + c_1$ | $a_1$ | 1 | $tmp$ | 8 |
| | $c_1$ | 5 | | |
| 8: $out_2 = tmp$ | $tmp$ | 7 | $out_2$ | 0 |

Table 3.1: The Use/Definition table of the AR filter example.

| *range* | op | val | die[3] |
|---|---|---|---|
| 0 | *begin* | $in_1$ | 2 |
| | | $in_2$ | 3 |
| | | $p_1$ | 1 |
| | | $p_2$ | 2 |
| | | $p_3$ | 4 |
| | | $p_4$ | 5 |
| 1 | * | $a_1$ | 6 |
| 2 | * | $a_2$ | 3 |
| 3 | + | $b_1$ | 4 |
| 4 | * | $out_1$ | 7 |
| 5 | * | $c_1$ | 6 |
| 6 | + | $out_2$ | 7 |
| 7 | *end* | | |

Table 3.2: The Binding table of the AR filter example.

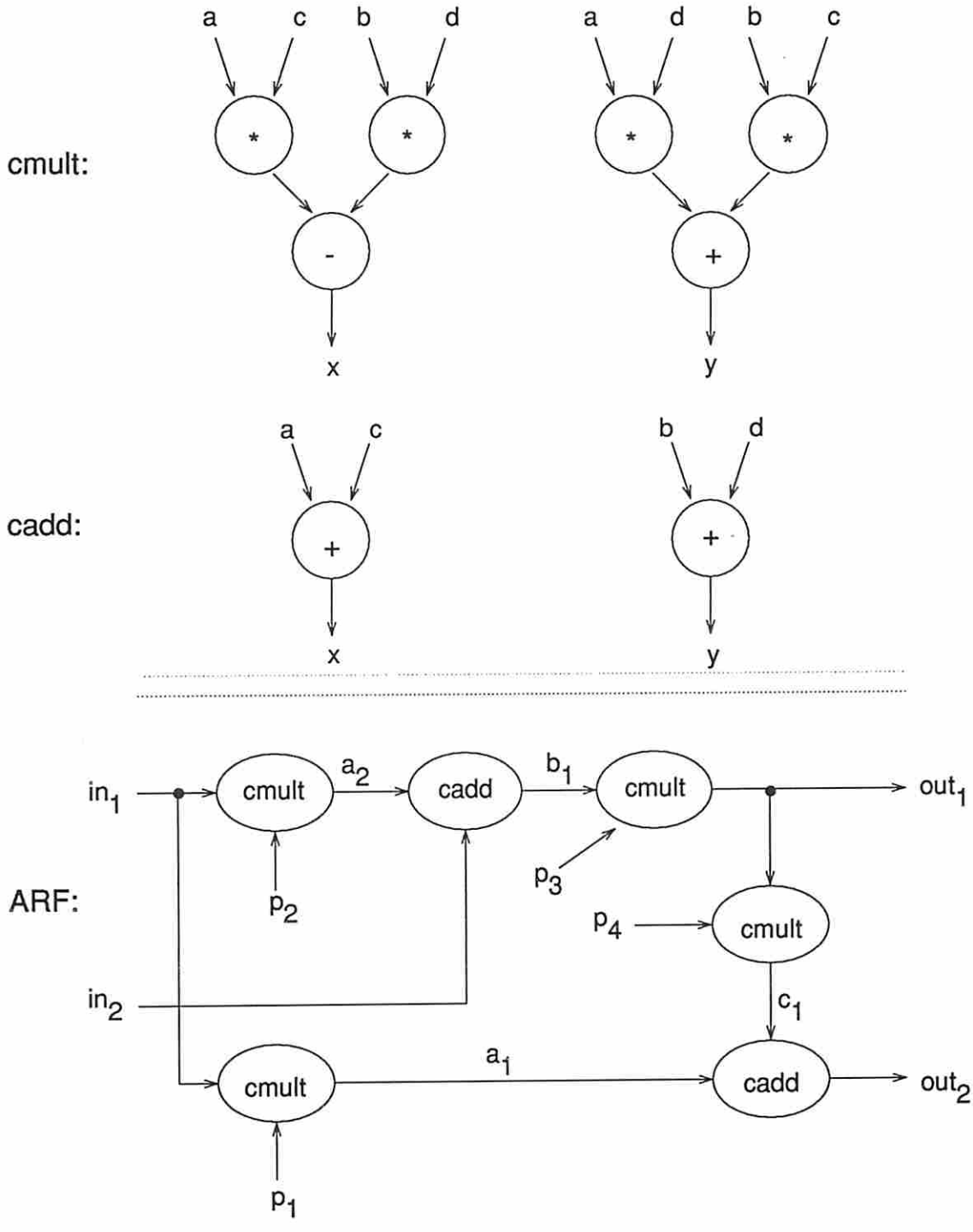



Figure 3.3: The data flow graphs of the AR filter example.

# Chapter 4

# Conclusions

In this document, we have introduced the VHDL2DDS program which is part of the USC ADAM system. The underlying VHDL subset used was defined and the corresponding DDS representations were also given. In addition, the analysis techniques used in the translation process have been briefly described.

We believe the principal concepts developed for VHDL2DDS could be applied to translate general high-level languages to a data flow language [3] which is more suited for today's highly parallel machine. This application is valuable because the inherent parallelism in a conventional program can be exploited by this kind of translation without resort to the parallel constructs, e.g. *parbegin/parend*, or recoding using a special parallel language.

Several enhancements to VHDL2DDS could be made to improve the usability and the capacity. VHDL2DDS currently is only a prototype program whose primarily goal is to demonstrate an approach to translate the VHDL descriptions into the DDS representations. Thus, very little optimization has been done. However, prior to the synthesis, there is plenty of scope to optimize the input specifications because of the use of high-level constructs. The main objective of the high-level optimization is to generate optimized behavioral designs for the synthesis system so that hopefully better structural designs can be obtained after the synthesis. In order to make VHDL2DDS more usable, the VHDL parser could be implemented in VHDL2DDS instead of using a separate program, and a much larger VHDL subset should be used (at least the structural constructs must be allowed) to provide better expressive power for the design specification.

# Appendix A

# Naming Convention

This appendix describes the naming convention used by VHDL2DDS to name its DDS output. Currently, the output of VHDL2DDS is directed to ADAM's DDS database. This database is an object-oriented database where every object has an unique identifier. However, only a global naming space is provided for all objects in the database, and the length of an object identifier is limited to 25 characters. Therefore, hierarchical naming and identifier scoping is not possible unless some *mapping* mechanism or naming convention is used. Since the interactive access to database objects is required, the naming convention approach is used.

## A.1 The Naming Convention

In DDS, the fundamental structure is the *component*. Each component is described in terms of four *models*, the *dataflow*, *control-timing*, *structural*, and *physical* models. Hence, it is natural to divide the single DDS naming space into several subspaces based on the components. All objects related to a component share the same naming subspace. The naming template is given below:

<*component identifier*>[<*model flag*>[<*model-specific object identifier*>]]

i.e., each object identifier may consist of three parts: the component identifier that indicates the particular naming subspace, the model flag that denotes one of the four models, and the model-specific object identifier.

1. *Component Identifiers*

   The component identifier is derived directly from the identifier of an entity, a package, a subprogram, or an operator in VHDL. Identifiers longer than 5 characters are truncated. A component identifier is recursively made by the following naming template:

   [<*the component identifier of the enclosing context*>]\ <*the truncated identifier*>

   e.g., if a subprogram named 'SUB' is declared within package 'PACK', the corresponding component identifier of the subprogram will be 'PACK\SUB'.

2. *Model Flags*

   The model flag is a special character used to denote one of the four models. Currently, the model flags for the dataflow model and the control-timing model are '.' and '#'. For example, the control-timing model of the subprogram given in previous item will be named as 'PACK\SUB#'. In addition, '$' is used as the flag for operation binding.

3. *Model-Specific Object Identifiers*

   (a) *Data Flow Model*

   There are three object types within this model. They are *value definition*, *value reference*, and *operation reference*. The *operation definition* is the data flow model itself. The object identifier of a value definition is derived from the corresponding type name in VHDL, and the type name is truncated as well. For example, the type BOOLEAN in package STANDARD is named 'STAND.BOOLE'.

   If the value reference is an interface value for a given component, its object identifier is the truncated name of the corresponding interface object in VHDL; otherwise, it will be given an internal value number in '∧V<*number*>' format. The associated net if present is named by appending ':N' to the object identifier of the value reference.

   All operation references within a dataflow model are given an internal operation number using '∧O<*number*>'. The pins of an operation reference are numbered and named by appending ':P<*number*>' to the object identifier of the operation reference.

   (b) *Control-Timing Model*

   The control-timing model is actually a *range definition*. In this model, all *point definitions* are primitives and are given a predefined name[1]. Both *range references* and *point references* are numbered and their object identifiers are '∧R<*number*>' and '∧P<*number*>' respectively.

   (c) *Operation Bindings*

   An operation binding represent a relationship between a dataflow element, a time range, and a structural element. In VHDL2DDS, only the the dataflow part and the control-timing part are given to an operation binding. Operation bindings are also numbered, and the identifier format is '∧B<*number*>'.

## A.2 Predefined Names

There are some constructs which are unique to DDS and don't have any counterpart in VHDL. Their names, therefore, are given by VHDL2DDS. The following is a list of the object identifiers of these constructs in the DDS database.

[1]See Section A.2 for the complete list of predefined names.

| *DDS Object Identifier* | *Description* |
|---|---|
| STAND\D | *distribute* operator |
| STAND\J | *join* operator |
| STAND#SIMPLE | *simple* point definition |
| STAND#ALPHA | $\alpha$ point definition |
| STAND#OMEGA | $\omega$ point definition |
| STAND#ORFORK | *or-fork* point definition |
| STAND#ORJOIN | *or-join* point definition |
| STAND#ANDFORK | *and-fork* point definition |
| STAND#ANDJOIN | *and-join* point definition |

# Appendix B

# Error Messages

The following error messages may be emitted by VHDL2DDS. The severity of the error can be either "Error" for detected violations of the VHDL2DDS's semantic rules, or "Fatal" for error conditions from which it is not possible to continue. The error messages are listed in the alphabetical order and their text is mostly self-explanatory. The error text may contain the string %s which indicate that the string is replaced by some location-specific text.

## B.1 Error-Level Error Messages

"%s not allowed for discrete range specification"
"%s not allowed for iteration scheme"
"%s not allowed for specifying the aliased object"
"%s not allowed in a static expression"
"%s not allowed in a static operation"
"%s not allowed"
"%s operator not allowed in a static operation"
"%s type definition not allowed"
"after clause not allowed"
"aliasing an alias object is not allowed"
"array dimension too large"
"bad index expression %s"
"bad l-value"
"bad r-value"
"concurrent statement %s not allowed"
"constraint not allowed in type indication"
"declaration type %s not allowed"
"illegal data type for index definition"
"illegal data type for iteration scheme"
"label not allowed in exit statement"
"label not allowed in next statement"
"more than 1 choice for a case alternative"
"more than one process in architecture body %s"

"multi-element waveform not allowed"
"no process in architecture body %s"
"process sensitivity list not allowed"
"resolution function not allowed"
"statement part of entity %s not empty"
"statement type %s not allowed"
"subtype declaration not allowed"
"transport delay not allowed"
"unreachable code"

## B.2 Fatal-Level Error Messages

"bad argument %s"
"can't create object %s"
"can't open DDS database %s"
"can't open DLS library %s"
"can't open DLS unit %s"
"can't open log file %s"
"can't open rec file %s"
"can't open sta file %s"
"can't relate %s to %s"
"illegal option %s"
"illegal view type %s"
"internal bug of vhdl2dds %s"
"no DDS database given"
"primary unit %s not translated"
"unit %s has been translated"
"unit type %s not allowed"

# Appendix C

# Known Bugs and Limitations

The following bugs or limitations are known to be present in VHDL2DDS version 1.7. Where possible, a workaround is given:

- If a subprogram is declared to be the interface of a module in the library, its formal parameters must be named 'OPN1', 'OPN2', $\cdots$, etc., in order to facilitate the module bindings.
- Occasionally, VHDL2DDS will generate unequal numbers of distribute operations and join operations in a DFG, which is not acceptable by the current coloring algorithm. This is due to the limited expressive power of the distribute/join mechanism and the naive rules used by VHDL2DDS to identify the conditional dependent values.

  *Workaround*: If a variable is used by both the then body and the else body of a conditional statement, and not redefined by either the then body or the else body, add a copy statement like $x := x$ after the conditional statement if it is no longer being used afterward. In general, a new coloring technique should be developed to color the DFG according to the CTG and bindings.
- The identifiers of interface objects, types, subprograms, packages, or design entities are truncated after the translation. It is possible to have naming conflicts in the DDS database if identifiers with a same 5-character prefix are used.

  *Workaround*: Don't declare identifiers having a same prefix longer than 5 characters within a declarative region.

# Bibliography


[1] W. B. Ackerman and J. B. Dennis. " VAL - A value oriented algorithmic language," Lab. Comput. Sci., Massachusetts Inst. Technology, 1978.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] S. J. Allan and A. E. Oldehoeft. "A flow analysis procedure for the translation of high-level languages to a data flow language," *IEEE Trans. on Computers*, pp. 826-831, Sep. 1980.

[4] F. E. Allen. "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, pp. 1-19, Jul. 1970.

[5] F. E. Allen and J. Cocke. "A program data flow analysis procedure," *Comm. ACM*, vol. 19, pp. 137-147, Mar. 1976.

[6] Arvind, K. P. Gostelow, and W. Plouffe. "An asynchronous programming language and computing machine," TR-114a, Dep. Comput. Sci., Univ. of California, Irvine, Dec. 1978.

[7] Arvind and K. P. Gostelow. "The U-Interpreter," *IEEE Computer*, pp. 42-49, Feb. 1982.

[8] C. T. Chen. "The Translation of VHDL Language to DDS Data Structure," Dept. of Elect. Engi. - Systems, Univ. of Southern California, 1990.

[9] C. T. Chen. "Manual Pages for the VHDL Language Interface," Dept. of Elect. Engi. - Systems, Univ. of Southern California, 1990.

[10] *Design Library System*, CAD Language Systems Inc, 1989.

[11] *VHDL Analyzer User's Manual for Sun/SunOS*, CAD Language Systems Inc, 1989.

[12] J. B. Dennis and K. S. Weng. "First version of a data flow procedure language," *Programming Symp.: Proc. Colloque sur la Programmation*, pp. 362-376, Springer-Verlag, Apr. 1974.

[13] M. S. Hecht and J. D. Ullman. "A simple algorithm for global data flow analysis problems," *SIAM J. Comput.*, vol. 4., pp. 519-532, Apr. 1975.

[14] *IEEE Standard VHDL Language Reference Manual*, The Inst. of Electrical and Electronics Engineers Inc, 1988.

[15] R. Jain, K. Kucukcakar, M. J. Mlinar, and A. C. Parker. " Experience with the ADAM Synthesis System," *Proc. of the 26th Design Automation Conf.*, ACM/IEEE, Nov. 1988.

[16] D. Knapp and A. C. Parker. " A Unified Representation for Design Information," *Proc. of the IFIP Conf. on Hardware Description Languages*, Aug. 1985.

[17] M. McFarland and A. C. Parker. "An abstract model of behavior for hardware description," *IEEE Trans. on Computers*, vol. 32, pp. 621-637, Jul. 1983.

[18] M. Mlinar and A. C. Parker. "Loop Transformations for Acyclic Data Path Synthesis," Tech. Report, Dept. of Elect. Engi. - Systems, Univ. of Southern Cal., May 1989.