A Design Representation Model For High-Level Synthesis

Elke A. Rundensteiner and Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
September, 1990

Technical Report 90-27

A DESIGN REPRESENTATION MODEL FOR HIGH-LEVEL SYNTHESIS

Elke A. Rundensteiner and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

September, 1990

ABSTRACT

Design tools share and exchange various types of information pertaining to the design. The identification of a uniform design representation to capture this information is essential for the development of a successful design environment. We have done an extensive study on the representation needs of existing database tools in the UCI CADLAB; examples of which are graph compilers for high-level hardware specifications, state schedulers, hardware allocators, and micro-architecture optimizers. The result of this study is the development of a design representation model that will serve as a common internal representation (DDM) for all system and behavioral synthesis tools. DDM thus builds the foundation for a CAD Framework in which design tools can communicate via operating on this common representation. The design information is composed of three separate graph models: the conceptual model, the behavioral model and the structural model. The conceptual model (represented by a Design Entity Graph) captures the overall organization of the design information, such as, versions and configurations. The behavioral model (represented by an Augmented Control/Data Flow Graph) describes the design behavior. The structural model (represented by an Annotated Component Graph) captures the hierarchical data path structure and its geometric information. In this paper, we define the last two graph models. They both capture the *actual design data* of the application domain. Since VHDL has gained increasing popularity as hardware description language for synthesis, we give numerous examples throughout this report that show how the proposed design representation model can be used to represent VHDL specifications.

Key Words: Design Data Model, Design Representation for Computer-Aided Design, Hierarchical Control/Data Flow Graph, State Transition Graph, Annotated Component Graph.

Contents

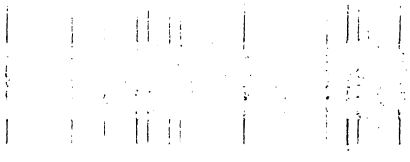
List of Figures

1 INTRODUCTION

Design tools, such as system and behavioral synthesis tools, have to share and exchange diverse types of information during the course of the design exploration process. For the integration of this variety of design information into one unified representation we define the *design data model* (DDM). The identification and definition of such a uniform design representation is essential as the synthesis tools interact via operating on this common representation. This design representation is to be maintained by a design database. In this report, we describe the design model, while a description of the design data base architecture and its functionality will be given in a later report.

High-level synthesis is concerned with the mapping of a behavioral specification written in a hardware description language to a structural description representing a set of interconnected generic components (i.e., a netlist). A description language is generally not amendable for direct translation into hardware. Hence, the behavioral description gets compiled into an internal representation that contains data flow and control flow information as implied by the specification. A well-defined design representation serves as a 'canonical' form into which different input formats can be mapped. For instance, different description languages, such as Ada or VHDL, can be mapped to the same internal representation. This makes design tools *language-independent* from characteristics of particular description languages. Furthermore, an internal flow graph representation allows for compiler-optimizations that would be hard to perform on a textual representation. Also, it is a flexible representation that can gradually be updated during synthesis.

The design model is composed of three separate graphs: the *conceptual graph model*, the *behavioral graph model* and the *structural graph model.* The first model, sometimes also called the *meta-data model*, describes the conceptual data schema that is used to organize the design data. It covers concepts, such as, design entities, versions, and configurations. This conceptual model serves as foundation for most database support functions, for example, version management, transaction processing, and schema browser. The other two models are *information models* which capture the *actual design data* of the application domain. They describe the design at a level at which the design tools are ultimately interested working on. We distinguish between the two domain types, the behavioral domain and the structural domain. The behavioral graph model describes the behavioral specification of the design. It corresponds to a hierarchical Control/Data Flow Graph (CDFG) representation that is augmented with timing constraints and state information. The structural graph model is a hierarchical graph structure of interconnected components augmented by



timing constraints (called Annotated Component Graph). It captures the data path structure and its geometric implementation, called the floorplan. In this paper, we present the extended Control/Data Flow Graph (CDFG) model and the Annotated Component Graph model, while the conceptual data model will be discussed in a later report.

The document is structured as follows. In Section 2, we present the foundation of our work. In particular, we discuss system and behavioral synthesis applications requirements for a design representation. The behavioral domain representation, the augmented CDFG, is described in Sections 3 to 5. Section 3 presents the data flow graph model, Section 4 defines the control flow graph model, and Section 5 describes how the state information is integrated into the CDFG model via the state transition graph model. The structural information domain, represented by the Annotated Component Graph, is defined in Section 6. Sections 3 through 6 contain examples that describe how the the design representation is used to model designs described by VHDL specifications. Section 7 summarizes our approach towards linking the behavioral and the structural information domain. Finally, in Section 8 we give conclusions.

2 THE DESIGN DATA MODEL: BASIC CONCEPTS

The success of a design data base for a particular application domain is to a large degree based on the quality of the underlying design representation, the design data model. This paper describes such a design data model, DDM, that is targeted towards supporting CAD applications. This model is intended to serve as an internal representation schema for a collection of behavioral synthesis and verification tools.

2.1 General Goals

The design representation we propose in this document has been designed with the following goals in mind:

1. The design representation must be general, i.e., language-independent, such that descriptions written in other specification languages can also be translated into the internal representation. This assumes that a suitable compiler is developed for each new language which compiles a description in this language into the internal representation.

2. Each design tool or group of related design tools generally keeps its own *internal data model*, i.e., its own format and data structures. One goal of this research is to achieve a consensus between these different *internal data models* by proposing a general design representation. This model needs to integrate design data as generated or required by all of the design tools. Therefore, it should be flexible enough to encompass the needs of all existing and possibly future design tools. In other words, we propose to establish a design representation standard.

3. The completeness of the representation is a necessary but not a sufficient characteristic for a successful data model. The initial textual specification itself is complete but nevertheless not useful for direct synthesis. This suggests the additional requirement that the design representation has to support the synthesis process. A measure of suitability for high-level synthesis is, for instance, the ease with which relevant information can be extracted from it.

4. We target the design data model towards the needs of design tools. Consequently, *human readability* of the representation is of secondary importance. The design data base will however provide design views which provide formats more suitable to the human designer, such as, a textual state transition table.

5. VHDL [28] is a hardware description language which recently has become the IEEE standard. Because of its increasing popularity, synthesis tools are being developed that synthesize from VHDL descriptions. Therefore, the proposed design representation should be powerful enough to represent all VHDL constructs that are useful for synthesis. We have studied VHDL in depth, and throughout this document we explain how the proposed data model can be used to represent VHDL specifications.

2.2 Multiple Domains

The design data model, DDM, has to integrate different information types into one unified representation. A design entity is originally just specified by a VHDL textual specification which then gradually gets transformed from a flow-graph representation over a structural description down to layout. Generally, synthesis systems distinguish between four domains, which are textual, behavioral, structural and layout (Figure 1). We simplify the diversity of domains by classifying them into two groups, the behavioral and the structural one. The first corresponds to the behavioral description over time and the second to the description of the structure that implements that behavior. The *behavioral* information domain comprises:

- the textual VHDL input specification that describes the function of the design,
- the flow-graph representation which captures the behavior over time but pertains no information about its implementation, and
- the state sequencing that shows the slicing of the behavior into states.

We represent this behavioral domain by an extended Control/Data Flow Graph model as discussed in the next section.

The *structural* information domain consists of:

- the data path structure that shows the decomposition of the design entity in terms of components and their interconnections, and
- the geometric information that describes the circuit's geometric layout but by itself conveys no information about its functionality.

The structural representations in the literature generally take the form of some annotated net list. Our structural representation, called the Annotated Component Graph (ACG), is similar. It is extended, however, to handle structural hierarchy. Furthermore, advanced attributes of the structural elements, such as, different delays for different inputs of unit, are supported. In addition, we directly associate geometric information with the ACG graph rather than creating a separate geometry representation graph [13].

2.3 The CDFG Model

There is no agreement on design representations of behavioral information for synthesis in the literature. Some of the commonly used approaches for design representations at the behavioral level are flow graphs, event graphs, and Petri nets. The representation presented in this report is based on the first approach. However, it includes essential elements from all three approaches into one unifying model.

Flow graphs are a common intermediate design representation for synthesis tools. They generally are based on a synchronous model and thus do not allow the representation of asynchronous events. In addition, most of these models do not address issues, such as, modeling of timing constraints, hierarchy, and concurrency. The data flow graph structure, such as, DSL by Camposano et al [6, 4], the data flow graph model used by Temme [27] are representative for the more popular approaches towards flow graph representations for synthesis. The DSL representation consists of a flat data flow graph augmented with control flow arcs to handle control constructs like branching and loops. Control flow has only been introduced as an afterthought as it is not a first class citizen of the specification. The control/data flow graph (CDFG) [20] represents a hybrid representation. Rather than modeling control flow by augmenting an existing data flow graph with sequence arcs, the CDFG model distinguishes between the control and the data flow portions of a description. The control flow graph of CDFG explicitly model the control constructs found in the original design specification rather than embedding them within the data flow graph.

In [26], the relative strengths and weaknesses of these two major approaches towards flow graph representations for synthesis are compared. It was found that a hybrid control flow/ data flow representation bears numerous advantages over a flat data flow graph model. It models the behavioral specification in a more direct manner by retaining the original structure of the input specification, since control flow nodes "call" data flow graphs. Thus, there is a one-to-one correspondence between the control flow graph and the structure of the design specification. This helps the

designer to visualize the flow of control, and thus provides a natural means of design entry, much like *flow-charting*.

The control flow graphs show necessary *sequencing* that is to be guaranteed by the control unit, whereas data flow graphs are designed to expose *maximum parallelism* in the input description. The exposition of maximal parallelism in a data flow graph helps to utilize the data path to its fullest capacity. Transformations can be designed to, for instance, map a conditional statement from control flow to data flow, or vice versa. If a statement is implemented in control flow then that the branching is executed by the control unit. If it is described in data flow then the condition is directly expressed in terms of data path units. This way the potential concurrency between the conditional and other data manipulation operations is revealed. This hybrid representation clearly allows for a data-flow/control-flow trade-off; it is thus suitable for synthesis. The data flow portions of the CDFG graph can be refined and finally mapped to circuits of components, while the control flow portions are reduced and finally transformed into the state table.

For these reasons, DDM has chosen a Control/Data Flow Graph (CDFG) model for the representation of the behavioral domain. In [16] and [26], it is shown how some of the VHDL constructs can be represented by the CDFG representation. In this document, we go beyond this work by augmenting the CDFG representation to also handle advanced concepts, such as, hierarchy, concurrency, events, timing constraints, and memories.

2.4 The Design Representation Continuum Problem

The initial input to the synthesis system is a pure behavioral description of the design, while the final output of the system is an optimized floorplan. There clearly is a design representation continuum ranging from a pure behavioral representation over many intermediate organizations to a geometric implementation. Ideally, we would like to directly model this continuum such that any intermediate state of the design can be retrieved. In practical terms this means that the design data model either

1. keeps many different snapshots of the design representation as it is gradually modified as well as links between objects of adjacent snapshots, or
2. the initial behavioral representation is gradually transformed into a structural one.

The second approach is likely to hide information because some transformations are not reversible. Hence, it will not allow us to go back to previous stages of the design evolution. Consequently, pure behavioral views may no longer be extractable from this modified representation. Also, relationships between behavior and structure that are needed for redesign may not be known.

The former approach does not face these problems. In addition, it suggests an increased ease of use since the related objects can be retrieved directly by linkage traversal. It is expensive in space and maintenance costs, however, as multiple distinct versions of the design with a complex network of linkage would result. The ADAM design environment at USC, for instance, maintains detailed linking relationships as described in numerous papers [3, 13]. The required maintenance was found to be substantial, and to our knowledge made the creation of a workable database system extremely difficult [15].

Clearly, a solution lies in between these two extremes. We adopt the following approach. We keep two domain graphs rather than four as done by most other synthesis systems. Secondly, the behavioral domain graph is augmented with structural correspondence without modifying the original shape of the flow graph. Whenever this is not possible, i.e., behavioral modifications are required to reflect the structure, then a new version of the behavioral graph is created.

An important goal of our research is to reduce the complex behavior-to-structure links from the abstract behavior down to the final structure (Figure 1). We explicitly represent links in the design model only when necessary. The types of semantic relationships that can be identified between the objects of the different domains are listed below.

- Connections from the textual description to the flow graph representation;
- Connections from control flow to data flow constructs;
- Connections from each state to the data flow constructs it contains;
- Connections from each state to the control flow constructs it corresponds to;
- Connections from a behavioral operator in the data flow graph to a functional unit in the data path;
- Connections from each state to the data path units that are performing operations in the state;
- Connections from the structural domain (data path units or control unit) to the geometric domain.

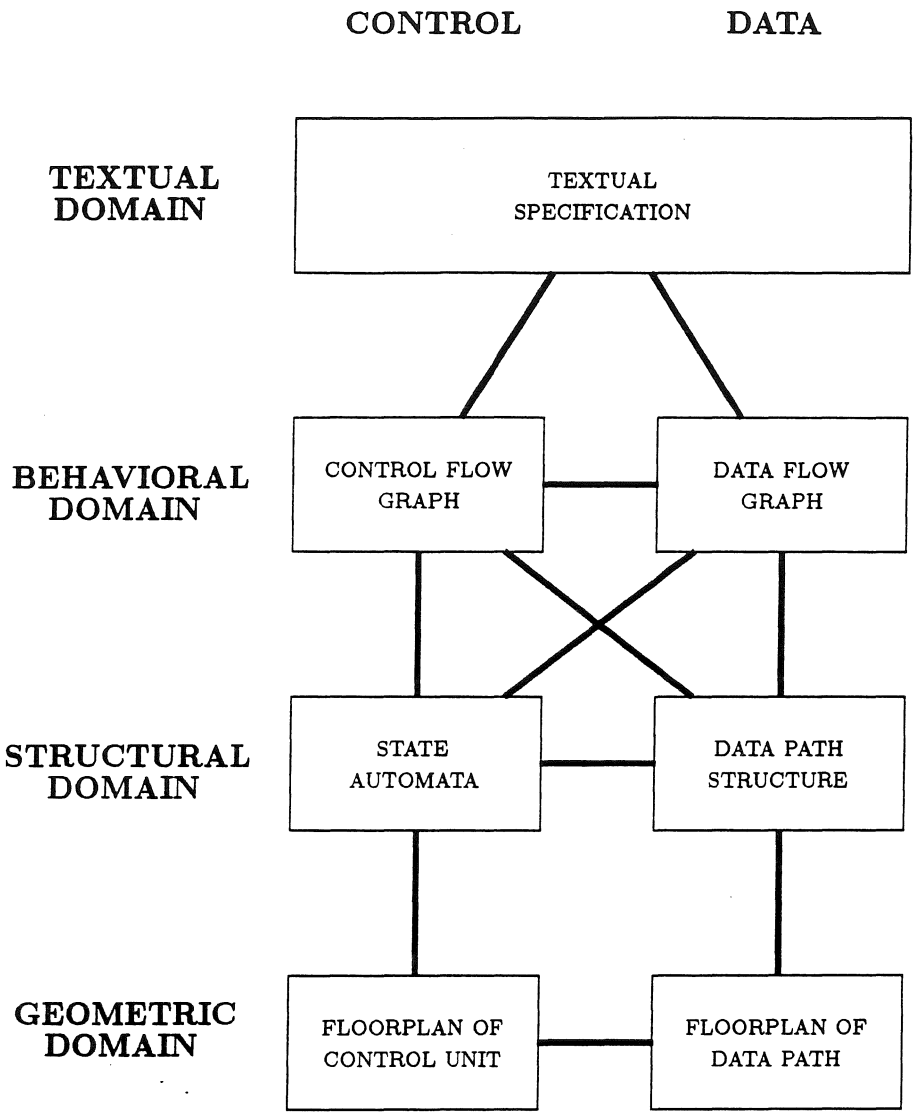


Figure 1: Possible Relationships Between Domains

The different representations must be linked together to keep track of the many-to-many relationships between the original design specification and the ultimately synthesized structure. Design linking between the control and the data parts is of importance for applications, such as, synthesis, debugging, verification and iterative design [4]. For instance, the control synthesis compiler needs to know which functionality of a unit is used in a given state. The problem of which relationships to represent has been studied in the literature. It is, however, still an open research question. Our approach towards this problem consists of combining the textual and the behavioral domain into one representation, and the structural and geometric into another one. Linkage across these two domain graphs is done by keeping structural information associated with the behavioral representation in the form of annotations. Motivation and explanations of our approach are given throughout the paper, and Figure 61 in Section 7 then summarizes our solution.

2.5 Foundation of DDM

We use a simple data modeling methodology to define DDM. The basic components of this method are given below. These terms will be used throughout the rest of the specification.

Object types are abstract type definitions that are defined by DDM. **Objects** are distinguishable entities of these object types. The actual design data then corresponds to a collection of such related **objects**. We distinguish between different groups of object types; those that form the **CDFG graph** and those that form the **ACG graph**.

Objects have an **object identifier** and a **state**. The object identifier is an identifier that can be used to uniquely refer to the object. The state consists of a collection of state variables, which can be either **attributes** or **relationships**. All state variables are named. The type of data referred to by the state variable is called the **domain**. **Attributes** describe characteristics of the object. The domains of attributes corresponds to primitive data types, such as, integer, string, as well as predefined enumeration types. An **attribute value** can be a single data value, a set of values, or an ordered list of values. **Relationships** between different objects are represented by references in the respective objects. Therefore, references are directed. There will generally be a "back reference" in the referenced object; which is used to make references symmetric. References themselves cannot carry any attributes. If we want to model an abstract relationship between two or more objects that further has to be described by additional attributes, then this relationship is represented by an explicit object definition. This object definition then not only holds the attributes of

the abstract relationship but also contains references to all objects that are related by this relationship type. In the following sections, we describe the different object types supported by DDM.

3 THE DATA FLOW GRAPH

DDM uses an augmented Control/Data Flow Graph (CDFG) model to represent the behavioral domain. The CDFG model distinguishes between the control and the data flow portions of a description. The data flow graph of the proposed CDFG model is discussed in this section, while the control flow graph is presented in the next section. A data flow graph is created for data manipulation operations, i.e., for assignment statements. Conditional statements can be represented both by data flow graph constructs or by control flow graph constructs. In the sequel, we first define the data flow graph and then we discuss its constructs in more detail. Thereafter, we also give numerous examples of how the data flow object types can be used to model different VHDL specifications.

3.1 The Data Flow Graph Definition

A data flow graph is defined as described below.

Definition 1 *A* **data flow graph** *is a directed (not necessarily acyclic) graph*[1] *DFG = (DN,DV,DE,DP,DF) with DN and DV the set of vertices, DE the set of edges, DP the set of ports, and DF the set of data flow marking functions. The elements of DN and DV are uniquely identified by the function* **vertex-num**: $\{ DN \cup DV \} \rightarrow INTEGER$.

1. *DN corresponds to the set of data flow nodes. It is composed of several disjoint sets,* $DN = OPERATION \cup FUNCTION \cup STORAGE \cup MARKER \cup TIME$.
   - *OPERATION is the set of all computation and selection operators.*
   - *FUNCTION corresponds to the set of function call nodes.*
   - *STORAGE corresponds to the set of variable accesses, array accesses and constants,*
   - *MARKER corresponds to the set of demarcation nodes.*
   - *TIME corresponds to the set of timing constraints (delays) in the data flow graph.*

[1]The data flow graph follows the single-assignment model, and therefore it will initially after graph compilation be an acyclic graph. However, cycles may be created after graph optimization. For instance, the read and the write nodes of a signal may get merged into one data flow vertex labeled by the read/write operator in order to simplify the mapping of this node to one register.

2. *DV corresponds to the set of data values produced or consumed by elements of DN.*

3. *DP is the set of data flow ports. They correspond to the connection points of vertices with the arcs of the graph. The function* **port-class**: $P \rightarrow \{input\text{-}port, output\text{-}port\}$ *specifies whether a port is an input or an output port. A vertex* $v \in DV \cup DN$ *can have an ordered (possibly empty) list of input and output ports* $p \in DP$*, respectively. The function* **input**: $DN \cup DV \times INTEGER \rightarrow DP \cup \emptyset$ *is an assignment of input ports to vertices. The function* **output**: $DN \cup DV \times INTEGER \rightarrow DP \cup \emptyset$ *is an assignment of output ports to vertices.*

4. *DE represents the set of directed edges between the (ports of the) vertices of the data flow graph. The edges correspond to pairs (p1,p2)* $\in DP \times DP$ *with the direction of the edge from p1 to p2. We distinguish between five types of edges, which are data flow arcs, sequencing arcs, hierarchy arcs, timing arcs and demarcation arcs.*

5. *DF is a set of data flow marking functions* DF_i: $DN \cup DV \rightarrow DM$ *with DM the set of possible marks. These functions associate attribute values with the vertices of a data flow graph. Examples of such attribute functions are access-type, array-dimension, bit-width, just to name a few.*

The set of data flow nodes corresponds both to active elements of the design specification, that perform the data manipulations, and to passive elements, that represent storage elements, whose content is used and modified by the former. Examples of active elements are the arithmetic, logic and bit extraction data operations, the function calls, and the data selection operations. Examples of passive elements are variable references, array accesses, constants, and parameters of functions.

We treat data values as a separate conceptual entities that can have their own attributes and thus represent them by vertices (the set DV) rather than by edges. This decision simplifies the mapping between behavioral and structural information as will be discussed in a later section.

3.2 Representation of the Data Flow Vertices

The graphical depictions of the data flow graph vertices and edges are shown in Figure 2 and 3, respectively. The meaning of each vertex type is explained next. We also list some of their attributes.

Type: Operator node

models arbitrary user-defined operations, while the operator node corresponds to a predefined operation. Each function call refers to a particular function specification which is an encapsulated description of the function behavior. Thus each function node is connected via a demarcation arc to its respective function description.

The representation of parameter passing is resolved as follows. A function call node has zero or more data inputs that correspond to the actual parameters that are passed to the function as arguments. This is done in the order of occurrence, i.e., the data value connected to the first input port is passed to the first formal parameter, the data value connected to the second input port is passed to the second formal parameter, etc. The data output port of the function node corresponds to the return value of the function. Note that a function node creates *hierarchy* in the data flow graph as it encapsulates any arbitrary user-defined behavior by one function call node.

Type: Choose-value node

Graphic: Triangle with a choose value for each data input port

Description: A choose value models a conditional selection of one data value from a collection of two or more values. A choose value is used to model the conditional update of a variable, i.e., a choose value is created for each variable that is updated within a conditional statement.

A choose-value node has two or more data input ports and one or more control input ports. The nodes connected with the control input ports are called the condition variables. One choose value, also called condition guard, is associated with each data input port. These choose values are mutually exclusive constant values. They correspond to a sum of products of terms, where the number of terms in a product corresponds to the number of control input ports. Each term evaluates to a boolean constant for a binary branching decision point (such as, an if-statement) and a constant or a range of constants of an integer, boolean, or an enumeration data type or a special **don't care** or **else** symbol for a multi-branch decision point (for example, a case statement). The choose values are compared against the condition variables. If a choose value matches the values of the condition variables, then the data value that is connected to the respective data input port is propagated through the choose value node. A choose-value node has a single data output port, that corresponds to the data value that is passed through choose value node.

Type: Read node

Graphic: Rectangle with a triangle taken out of left hand side and labeled by the variable name

Description: A read node is created for each read access, i.e., for each variable access on the right-hand side of an assignment statement or in a conditional expression. A read access can be to a variable, a signal, or an external port. A read node have one control input port and two output ports. A demarcation arc from the demarcation node for the beginning of the data flow graph points to its input port. The data output port is connected to a node that represents the data value read. The control output port is connected via a sequencing arc to a write access node of the same variable, if there is any.

Type: Write node

Graphic: Rectangle with a triangle added onto the left hand side and labeled by the variable name.

Description: The write node is created for each write access, i.e., for each variable access on the left-hand side of an assignment statement. A write access can be to a variable, a signal, or an external port. A write node has two input ports and one output port. The data input port is connected to a vertex that represents the data value that is written. The second input port of type control is connected to a sequencing arc from a read node of the same variable. The control output port is connected via a demarcation arc to the demarcation node that marks the end of the graph. If the variable is of type register, then the write node may have additional input ports of type control. They represent control lines, like for instance, reset and enable lines. For variables that are of type register, read and write nodes can be combined into one node during an optimization phase. Such a read/write node is graphically represented by a rectangle with a triangle both added and deleted from the left hand side[2].

Type: Constant node

Graphic: Rectangle with constant value

Description: A constant node models constant values. A constant node has one control input port that is connected via a demarcation arc to the demarcation node that marks the beginning of the graph. A constant node has one data output port that represents the constant value read.

Type: Read-array node

Graphic: Rectangle with a triangle removed from the right hand side and labeled by the array name.

[2]This symbol corresponds to an overlay of the graphical symbols for a read and a write node.

Description: A read reference to an array is represented by a read-array node. A read-array node takes two types of inputs. The first are sequencing arcs from other subscript nodes and the second the array address. For each dimension of the array, there is a data flow edge from the root of the expression that calculates the index value to the subscript node. A read-array node takes two types of output arcs. They represent the value read from memory and the outgoing sequencing arcs. Sequencing arcs are used to connect together successive references to the same array to preserve partial execution order of memory references.

Type: Write-array node

Graphic: Rectangle with a triangle added onto the right hand side and labeled by the array name.

Description: A write reference to an array is represented by a write-array node. A subscript write-array node has three different types of inputs: sequencing arcs from other subscript nodes, data arcs that represent index values, and one data value that corresponds to the value that is to be stored. Its only output port is connected to a sequencing arc. A write-array node is generated for every variable-index write array operation and for the latest value of an array element prior to the next variable-index read array operation.

Type: Data value

Graphic: Small circle

Description: Data value vertices correspond to the data values that are either generated or consumed by other data flow nodes. They model data dependencies between other data flow nodes. Such a data value node is sometimes as called a net. A data value node has attributes, such as, bit width, and data representation.

Type: Event node

Graphic: Diamond

Description: An event node in the data flow graph tests for an event on a signal. For example, an event node may test whether a signal is rising, falling, or stable. An event node has one input and one output data port. Input to the event node is a data value node (that represents a signal) and output is a boolean data value node (that represents the result of this event test).

Type: Timing node

Graphic: Stop sign symbol

Description: A timing node models a timing constraint on portions of the data flow graph. A timing node has one or more incoming timing arcs and one outgoing timing arc. The incoming timing arcs connect the delay node with the sources of the delay while the outgoing timing arc points to the destination data value node. The timing node specifies a delay for the execution of all nodes in the data flow graph between the source nodes and the sink node of the delay. Optionally, the timing node may associate an attribute called event-type (which takes the values RISING, FALLING, and CHANGING) with its source and its sink nodes. In addition, it may give a delay value for the a minimal, nominal and maximal delay constraint, respectively.

A timing node models two types of timing constraints, which are **path delays** and **event-related** delays. A **path-delay** timing node models the time taken for the effect of a signal to propagate through a set of hardware units from one point of the hardware to another. A **path-delay** timing node is given the label **path-delay**, or short, **delay**. By default, the path delay node constitutes a timing constraint for all data flow nodes on the paths starting from the source nodes of the delay node and ending with the destination node. A delay node may have an optional attribute, called **path expression**, which describes selected portions of the data flow graph. If a **path expression** is given, then the path delay only refers to the subset of data flow graph described by the expression.

An **event-related delay node** captures timing relationships between the occurrences of possibly independent events. Examples of such event relationships are set-up and hold times. An event-related delay node is given the label **event-delay**, or short, **event**. Such an event-related delay node can also be used to model duration timing constraints.

Type: Data flow graph demarcation nodes

Graphic: Half-circles

Description: The demarcation node pair **begin-dfg** and **end-dfg** mark off the beginning and the end of a data flow graph. The read and write nodes of a data flow graph are attached by demarcation arcs to the **begin-dfg** and the **end-dfg** nodes, respectively.

3.3 Representation of the Data Flow Arcs

The graphical depictions of the data flow graph edges are shown in Figure 3. Each edge type is further explained next.

Type: Data flow arc

Graphic: Arrow

Description: A data flow arc connects two data flow vertices by associating an output port of the former with an input port of the later. Data flow arcs show the flow of data values through the graph. Thus, one of the two vertices will be of type node (from the set DN) while the other will be of type value (from the set DV). This construct has been introduced due to the fact that we model data values as nodes rather than as arcs.

Type: Sequencing arc

Graphic: Bold arrow

Description: Sequencing arcs are used for enforcing sequencing among variable or memory access, whenever pure data dependencies are not adequate in representing correct program semantics. These arcs preserve an execution order that is implied by the semantics of the behavioral description but not captured by data dependencies.

Type: Hierarchy arc

Graphic: Bold dashed arrow

Description: A hierarchy arc connects two levels of a hierarchy. In particular, it connects a function call node with the specification of the function that is being called. The later is a control flow node that is to be discussed in a later section.

Type: Demarcation arcs

Graphic: Dotted arrow

Description: Demarcation arcs are used to associate all read nodes and all write nodes for variables and arrays with the **begin-dfg** and **end-dfg** demarcation nodes, respectively. We also use these sequencing arcs to point from a function call node to the body of the function or procedure that is to be executed *before* execution of the current graph continues.

Type: Timing arc

Graphic: Dashed arrow

Description: A timing arc connects a timing constraint node with a data value vertex. The set of timing arcs associated with a timing constraint node thus marks the group of data flow nodes that are constrained by the timing constraint node.

3.4 Modeling with Data Flow Constructs

In this section, we show how the just presented constructs can be used to represent a design at the behavioral level. In particular, we will use specifications written in VHDL for demonstration purposes.

3.4.1 Separate Data Flow Nodes for Accesses to the Same Variable

The proposed DFG representation maintains separate nodes for the different uses of a variable. That is, data values are not assumed to reside in one fixed location as done in [4]. Instead relevant values are passed from one data flow block to another. This parallels the fact that different values are bound to a variable over time. In other words, the results of two read-references to the same variable are a function of time. The CDFG representation models this directly by providing separate STORAGE nodes for these different accesses.

During one state a signal may potentially be captured by a register whereas during another state its value may be mapped to a wire. Keeping different conceptual nodes for different usages of a variable simplifies the maintenance of this linkage information between the data flow graph and the data path on a state by state basis.

3.4.2 Data Types

Access to variables are typically mapped onto either registers or memories. Therefore, the CDFG model directly supports data values that can be mapped to single bits, bit vectors, or arrays of bit vectors. All other data types will have to be mapped to such structures during flow graph optimizations. Integer and scalar values are represented as a bit vector. Arrays are represented by arrays of bit strings.

3.4.3 Operator Nodes Revisited

Operators are classified as arithmetic, comparison, shift/rotate, logical, bit manipulation, and special-purpose. The standard set of arithmetic operators, which are ADDITION, SUBTRACTION, MULTIPLICATION, DIVISON, INCREMENT and DECREMENT are assumed. The inputs have to be of the same data type. They produce outputs of the same type as the inputs. Shift operators are SHR, SHL, SHR0, SHR1, SHL0, and SHL1. Examples of comparison operators are EQ, NEQ, LE, LT,

GE, GT and examples of logic operators are AND, OR, NOT, NAND, NOR, XOR. They work on a bit-by-bit fashion on the operands.

The bit manipulation operators are CONCAT and EXTRACT. The CONCAT operator node accepts two or more binary data inputs. It concatenates them into a bit string in the order of the input ports. The EXTRACT operator node accepts one (binary) data input and extracts a specified range of bits. The EXTRACT and CONCAT nodes are further discussed in Sections 3.4.4 and 3.4.5, where we discuss the representation of access to registers and memories.

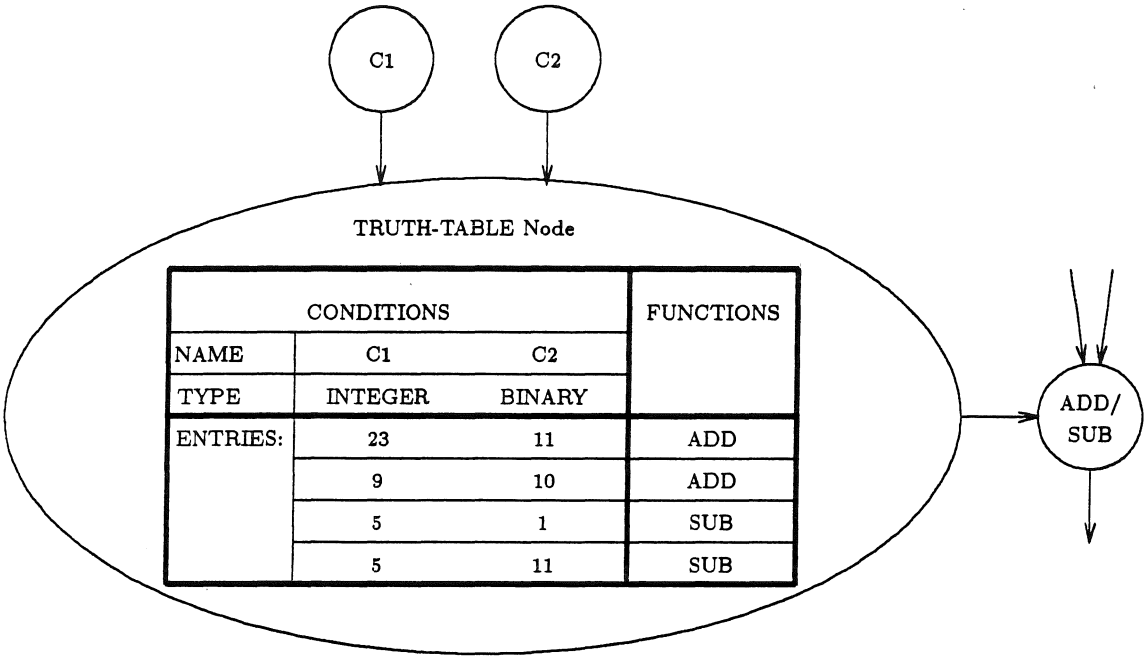


Figure 4: The TRUTH-TABLE Node Type

The special operator type is called TRUTH-TABLE node. Its behavior is described by a truth table rather than having predefined semantics. A TRUTH-TABLE operator node accepts one or more data inputs and produces one or more data outputs. It contains a truth table that translates the inputs to the outputs. The TRUTH-TABLE node type is not a direct translation of a behavioral description but it can for instance be used to capture the results of architectural optimizations on the representation [24, 25]. Figure 4 shows an example of a TRUTH-TABLE node. In this example, the node is used to encode the conditions for the function selection of a multi-functional operator node. The multi-functional operator node shown on the

right-hand side of the figure has two functions, ADD and SUBTRACT. Based on the values of C1 and C2, one of these two functions ADD and SUBTRACT will be selected. The function ADD is selected if ((C1=23 and C2=11) or (C1=9 and C2=10)), and the function SUBTRACT is selected if ((C1=5 and C2=1) or (C1=5 and C2=11)).

3.4.4 Access to Registers

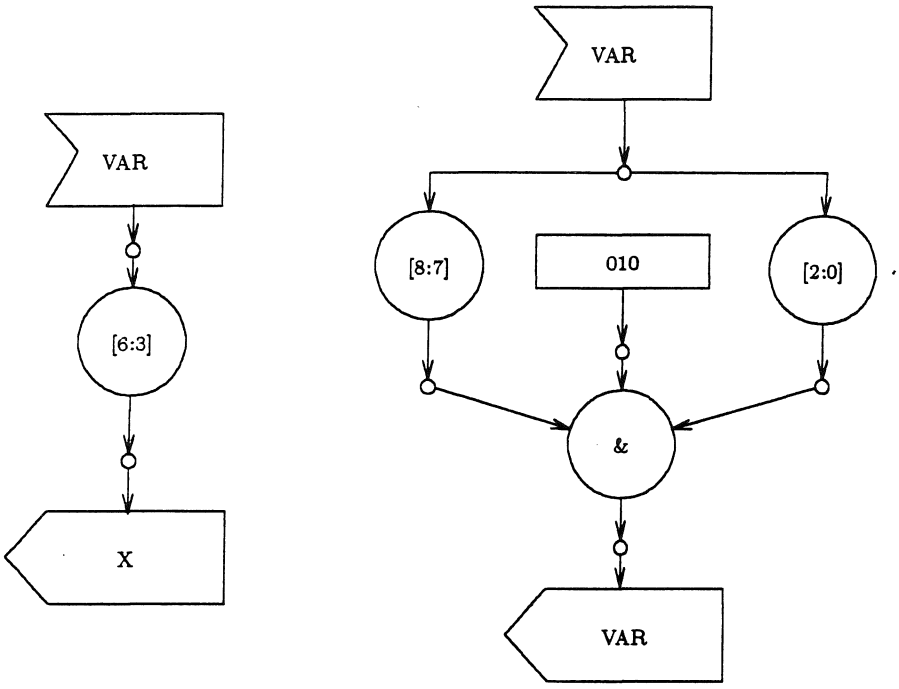
A register is modeled by a variable with an array of bits data type. For every bit-slice access to such a variable, the slice is constructed by concatenation and/or extraction operator nodes. A read access to a bit-slice of a variable is represented by an extraction node as is shown in Figure 5.a. A write access to a bit-slice of a variable is represented by one or more extraction nodes and a concatenation node. The latter constructs the new value of the variable.

For instance, the assignment VAR(6 downto 3) = "00" is translated into the statement VAR = VAR(8 downto 7) & "010" & VAR(3 downto 0) assuming that VAR is a bit-string with a range from 8 downto 0. In other words, a write access to a variable is modeled by reads to all portions of variable that are not to be modified by that write access. An example of this is shown in Figure 5.b.

3.4.5 Access to Memory

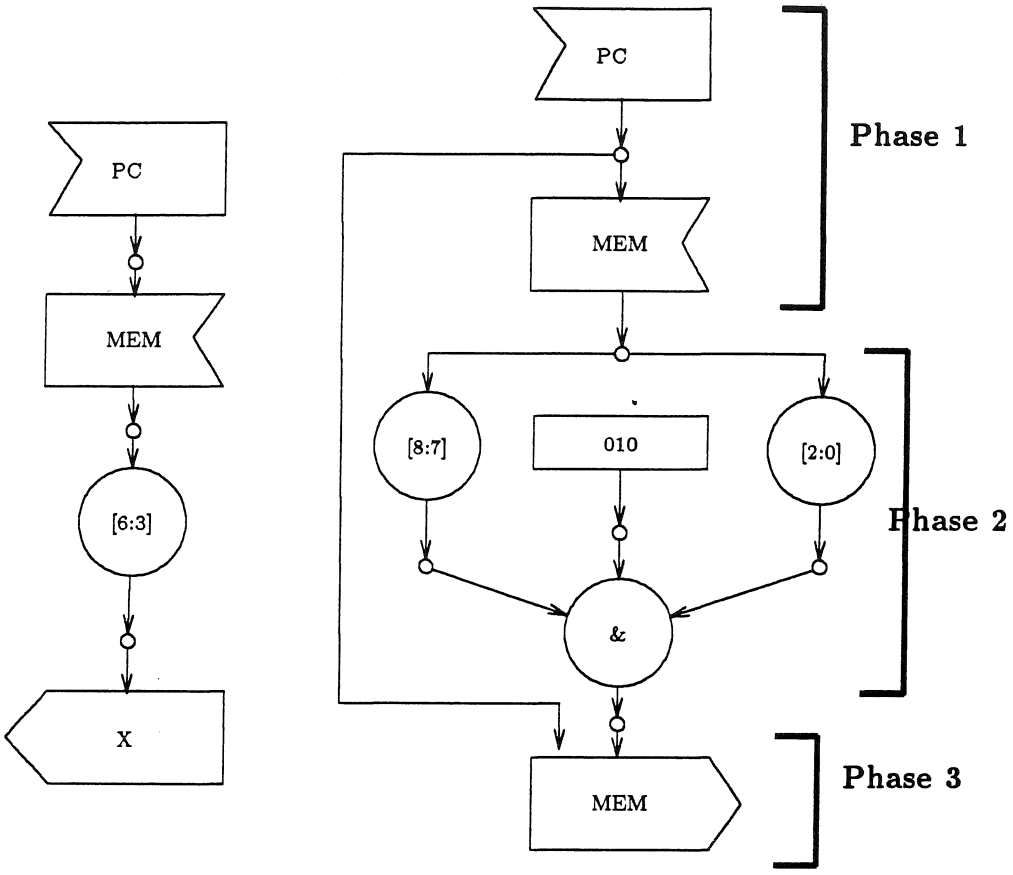
A memory unit is usually modeled by an array data structure where each element is of type bit vector. The CDFG model represents each textual reference to such an array by a subscript node. We assume that a memory unit can only be accessed sequentially, i.e., the only operations allowed on it are to read or to write one memory word. Consequently, only one index, which corresponds to the first dimension of the array, can be associated with each subscript node. Manipulations of the second dimension correspond to bit-slice operations on a single memory word. They are thus modeled by separate bit manipulation operator nodes independently from the array access, i.e., they are applied after the memory word has been extracted. This corresponds closely to reality: a memory access always reads or writes a complete word, and bit-slice operations are not performed within the memory but as separate operations.

For example, an array access that utilizes a bit-slice selection like the assignment X = MEM(PC,6 downto 3) is modeled in two phases. First it is represented by a data access of the array using the index for the first dimension, Y=MEM(PC), and then



a.) X = VAR(6 downto 3); b.) VAR(6 downto 3) = "010";

Figure 5: Variable Access Representation



a.) X=MEM(PC,6 downto 3); b.) MEM(PC,6 downto 3) = "010";

Figure 6: Subscript Access Representation

by a bit-slice selection operation on a single element of that array, X=Y(6 downto 3). The representation of this example is shown in Figure 6.a.

A write access to a memory location that modifies only some bits of the memory word is represented by a more complex sequence of operations. For instance, the assignment MEM(PC,6 downto 3) = "010" is modeled in three phases: first by a subscript read access Y=MEM(PC), then by a bit-slice selection and concatenation operation sequence, Y = Y(8 downto 7) & "010" & Y(3 downto 0), and lastly by the actual subscript write operation to the memory, MEM(PC)=Y. The design representation of this example is given in Figure 6.b.

3.4.6 Variable References and Dependencies

The sequential nature of typical hardware description languages imposes three types of data dependencies on the variable accesses. They are called **flow, anti** and **output** dependencies [21]:

- A **flow dependence** exists if a "Write S" is followed by a "Read S",
- an **antidependence** exists if a "Read S" is followed by a "Write S", and
- an **output dependence** exists if a "Write S" is followed by another "Write S".

In all three cases, inconsistencies will result if the given order of reads and writes is not preserved. No dependencies exist between a set of Read nodes if there is no Write node among them. These Read nodes referring to the same value of a variable are thus merged into one Read node.

If a **flow dependence** exists, then a sequencing arc is inserted from the Write node to the Read node of the same variable. This enforces that the value of the signal is updated before it can be read. Such a pair of consecutive Write and Read nodes may later be optimized by a graph critic. If an **antidependence** exists, i.e., a "Read signal" is followed by a "Write signal" node, then a sequencing arc is inserted from the Read node to the Write node of the same variable. This enforces that the value of the signal is read before it is allowed to be updated. If an **output dependence** exists and there is at least one Read signal statement between these two Writes, then no additional arcs are inserted. If there is no read access in between these two writes, then a sequencing arc has to be inserted between them to guarantee the correct output value for the signal at the end of the data flow block. The first write node can generally be eliminated since it is a dead-end operator. Whenever one of these

three data dependencies is already expressed indirectly by data flow edges, they do not have to be maintained explicitly.

3.4.7 Array References and Dependencies

The previous discussion about preserving the program semantics becomes even more intricate when array references are involved. The reason for this is that at compile time it is not always obvious whether the same memory location is accessed or overwritten for array references with variable indices. Hence, sequencing arcs are imposed between subscript nodes to the same array. If two indexes are constant values, then it can be determined immediately whether the sequencing arc between them is redundant. If one or both indexes are variables then dependency analysis techniques can sometimes be used to determine whether the arc is redundant. Once a sequence arc is found to be redundant, a flow graph optimization routine can remove this arc.

If a memory location is subsequently overwritten, then it may sometimes be possible to forego the generation of memory references nodes. One can use the data flow arc of the latest value of the array element instead of generating a separate access node. If a variable-indexed access write array node is encountered, subscript nodes have to be created for all prior accesses to that array and sequencing edges connect them with the new variable-indexed node. Sequencing arcs are always inserted between a variable-indexed write-array node and a subsequent array operation or between a write-array and a subsequent variable-indexed array operation of the same array. For example, sequencing arcs are inserted for the specifications given below:

if "A[I]:=X" is followed by "Y:=A[2]", or

if "A[I]:=X" is followed by "A[2]:=Y"; and,

if "A[2]:=X" is followed by "Y:=A[I]", or,

if "A[2]:=X" is followed by "A[I]:=Y".

3.4.8 Formal Procedure Parameters and Dependencies

Sequencing arcs are also used to maintain a strict ordering of the accesses to all parameters or ports of type **out** or **inout** with the same data type. This has to be done even if the parameters have different variable names and thus appear to be unrelated. The reason for this is the following. Formal parameters may be aliased to the same variable by assigning one actual parameter to two distinct formal parameters. Data

dependencies, that are not shown via data flow arcs in the representation, may thus exist between accesses to formal parameters.

For a given procedure call, the aliasing can be resolved and sequencing arcs of the design representation can be optimized. This implies that a separate copy is made of the procedure specification for each given procedure call, which then is synthesized to a specialized hardware. Below, we present one example to explain this situation.

In Figure 7.a, we show the design representation of a procedure specification body defined as follows:

```
procedure P (X: Type1, Y: Type1) is
begin
  X = I * 2;
  K = Y + 3;
  Y = X - I;
end;
```

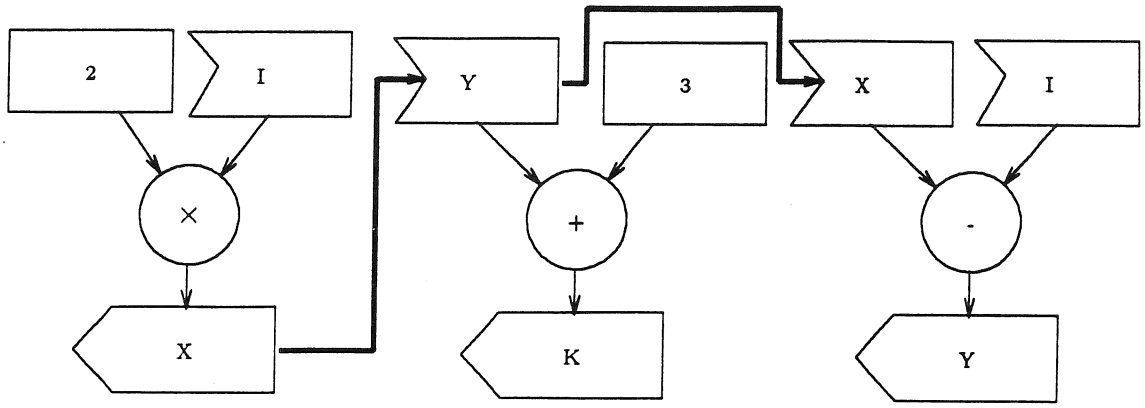
Sequencing arcs are maintained between the two formal parameters X and Y. In particular, the read node for parameter Y in the second statement is sequenced with the write node for parameter X in the first statement. Similarly, the read node for parameter X in the third statement is sequenced with the read node for parameter Y in the second statement.

If the two formal parameters X and Y are aliased to the same actual parameter then the flow graph can be optimized as shown in Figure 7.b. This would, for instance, happen if the procedure S is called in the following manner:

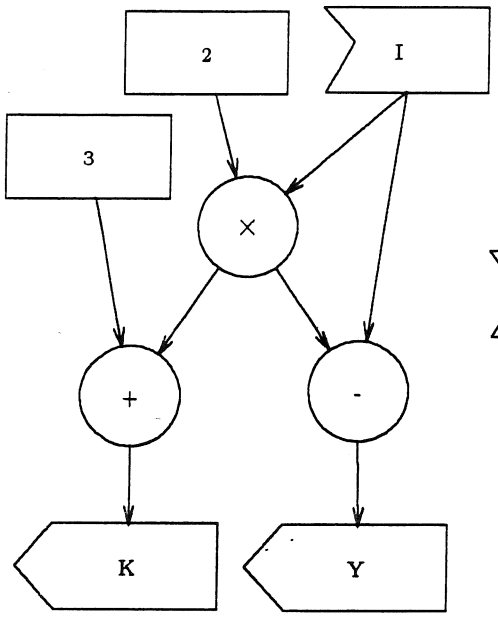
```
call P (VAR1,VAR1);
```

Then, the formal parameters X and Y refer to the same variable VAR1. Therefore, the sequence arcs preserving ordering between X and Y (Figure 7.a) are transformed into data flow arcs. They model actual data dependencies among occurrences of the variable VAR1. The optimized design representation corresponds to the following rewritten specification:

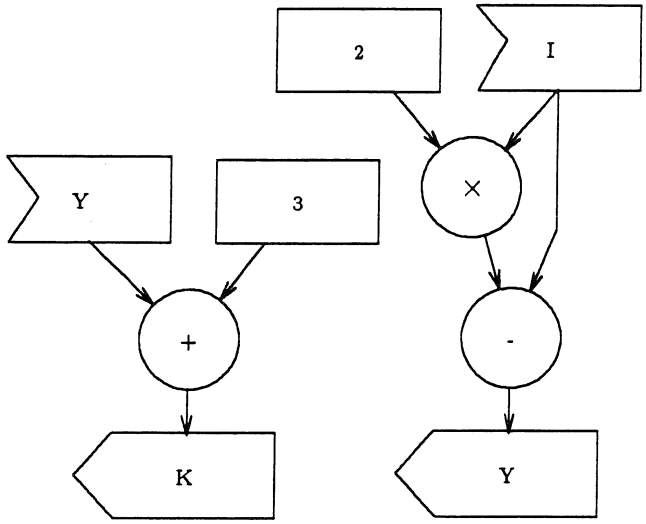
```
procedure P (X: Type1, Y: Type1) is
begin
  K = (I * 2) + 3;
  Y = (I * 2) - I;
end;
```



a) Original subprogram body.



b) Optimized Subprogram body after (X≡Y).



c) Optimized subprogram body after (X≠Y).

Figure 7: Formal Procedure Parameters and Dependencies

On the other hand, the formal parameters X and Y can be bound to distinct actual parameters by a procedure call described below, assuming that the variable VAR1 is distinct from variable VAR2.

```
call P (VAR1,VAR2);
```

Note that once it has been determined that X and Y are distinct, the specification could be rewritten as follows:

```
procedure P (X: Type1, Y: Type1) is
begin}
  K = Y + 3;
  Y = (I * 2) - I;
end;
```

Then the flow graph can be optimized as shown in Figure 7.c. In this case, the sequence arcs between X and Y can be removed, since X and Y are independent. This allows statement 2 to be executed independently and thus concurrently with statement 1 and statement 3.

3.4.9 Design Entity Ports and Dependencies

A strict order of all operations that deal with **input** and **output ports** of the design entity has to be maintained. Ports model communication points between different processors, i.e., data can be read from or written to these ports from outside the design entity. Thus, they may signify some prespecified behavior to the external environment, i.e., a fixed communication protocol between the design entity and the rest of the environment. Consider, for instance, the following statements:

localvar1 <= PortIn;

localvar2 <= fct (localvar1);

PortOut <= High;

with PortIn and PortOut input and output ports, respectively. Here, the last statement is not related by data dependencies with the other two. Consequently, it might be executed concurrently with the others. However, setting PortOut to High may be a sign for the environment that the value of PortIn has been read and thus

is free to be modified by other processes. This example shows the need for either a conservative approach towards exploiting parallelism from the computations of one processor, when fixed protocols for communication between design entities are not known, or for explicitly specifying these protocols.

We prefer the second approach. We assume that the designer explicitly specifies such protocols, if s/he wants to ensure a desired ordering of events. In Section 3.4.16, we introduce a mechanism to specify as well as represent event relationships, called the event-related timing constraint. An event-related constraint could for instance be inserted between the events of reading from PortIn and of writing to PortOut. This event-related constraint determines not only the ordering of events but also constraints on the duration of the time interval between the occurrence of two events. For a more detailed discussion of this construct see Section 3.4.16.

3.4.10 Modeling a Condition in the Data Flow Graph

Below, we discuss the construct used to model conditions at the data flow graph level, namely, the choose value node. In Sections ?? and ??, we then describe how this construct is used to represent different conditional statements.

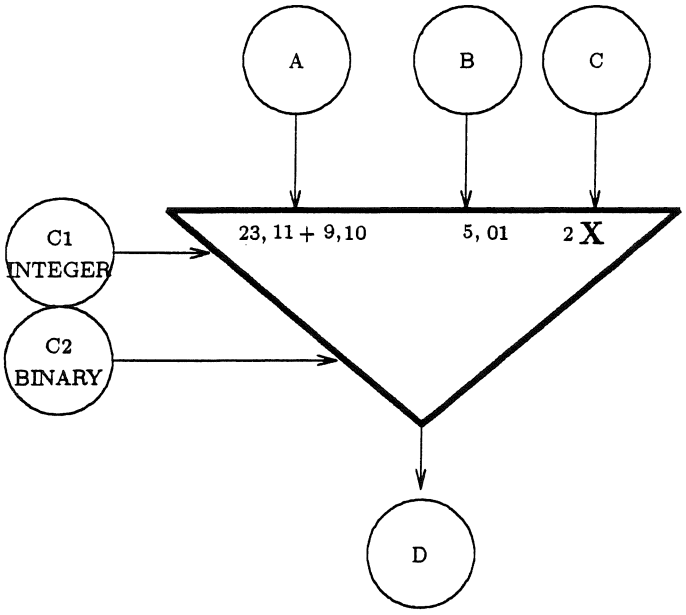


Figure 8: The Choose Value Node Type

The DDM model provides a generalized **choose value** node type in order to support design optimizations on the design representation. A choose value node models a conditional selection of one data value from a collection of two or more values. The mutually exclusive choose values associated with the data input ports of a choose value node have the following characteristics:

- they are composed of one or more products (or-ed),
- each product corresponds to a list of constant terms (and-ed); one for each condition variable attached to the control input ports of the choose value node,
- each constant term is either a constant value from a discrete domain corresponding to the data type specified for the condition variable, a **don't care** symbol or an **others** symbol.

This is best explained with an example. In the example in Figure 8, the choose value node selects among three different input data values, A, B, and C, which are connected to its three data input ports. Mutually exclusive choose values are associated with these input ports. The choose value for instance associated with the first data input port is "(23,11) + (9,10)". This choose value is a sum of two products, namely, (23,11) and (9,10). The number of terms in each product corresponds to the number of control input ports, in this case, it is two. The choose values are compared against the condition variables C1 and C2. The condition associated with the first data input port thus is evaluated as follows: ((C1=23 and C2=11) or (C1=9 and C2=10)). If this condition evaluates to **true** then the value of node A would be passed through the choose value node to node D. The choose value for the third data input port is "(2,X)". This interpreted to mean (C1=2 and C2=**don't care**).

3.4.11 Selected Signal Assignment Statements

```
with <expr> select
 signal  <=
         <waveform1> when v1,
         <waveform2> when v2,
         ...
         <waveformn> when vn;
```

Figure 9: VHDL Selected Signal Assignment

In VHDL dataflow style, a case statement is expressed by a selected signal assignment statement. An example of such a statement is given in Figure 9. The different values v1 to vn are mutually exclusive. If a value vi matches the select expression <expr>, then its corresponding waveform <waveformi> is evaluated and assigned to the signal.

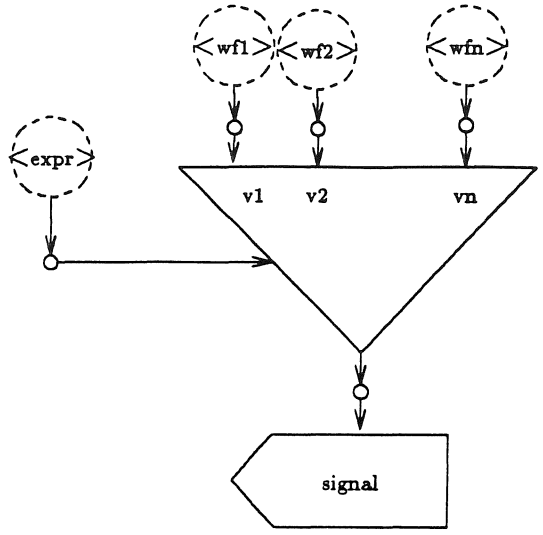


Figure 10: CDFG Choose-Value Node

The select expression <expr> can be a complex expression. Its only restriction is that it results in a discrete value. In Figure 10 we show how this statement would be represented in the data flow graph. The select expression <expr> is represented by a possibly complex data flow graph within which each access of a variable or array is represented by a **read** or a **read-array** node. The same is true for the waveforms <waveform1> to <waveformn>, i.e, each has a data flow graph generated for its expression value. One of them is selected to be assigned to the signal if its associated condition value v_i matches the value of the select expression. The signal on the left-hand side of the assignment is represented by a **write** or a **write-array** node.

In VHDL, a selected signal assignment statement can also be guarded. This means that the conditional signal assignment is based on the evaluation of the guard expression. The guard expression appears at the beginning of the enclosing VHDL block. It can be associated with any statement within that block. When the guard expression evaluates to **true**, then all signal assignments with a guarded qualifier appearing in this block will be evaluated.

```
block ( < guard expression > )
begin
with <expr> select
 signal   <= guarded
          <waveform1> when v1,
          <waveform2> when v2,
          ...
          <waveformn> when vn;
end block;
```

Figure 11: VHDL Selected Signal Assignment with Guard

In Figure 12 we show how the guarded select signal assignment shown in Figure 11 is represented in the CDFG model. A data flow graph (with **read** nodes) is generated for the guard expression of the block. A second choose-value node is added guarded by this expression graph. Its **true** input is connected to the result of the selected signal assignment (which can be directly duplicated from Figure 10), and its **false** input is connected with a read node of the signal.

Note that the guard expression is associated with the entire VHDL block, and thus other signal assignment statements within this block may also use it. This guard expression is thus represented once in data flow format and then it is directly linked to all choose-value nodes that guard a signal assignment within that block.

3.4.12 Conditional Signal Assignment Statements

An if-statement is represented in VHDL dataflow style by the conditional signal assignment statement. The syntax of this statement is shown in Figure 13. This assignment corresponds to a possibly nested if-then-else statement where all assignments are made to the same signal based on different boolean conditions.

Like conventional programming languages, the conditions <cond$_i$> are evaluated in the order in which they appear in the code. The first condition that evaluates to **true** is the only one to be executed. For instance, if the condition <cond$_j$> evaluates to **true** then its associated waveform value <waveform$_j$ > is assigned to the signal. If the statement is guarded then the additional condition that the guard evaluates to true has to hold.

This nested if-construct is represented by a chain of choose-value nodes as shown in Figure 14 [16]. For each if-statement a choose-value node is created guarded by

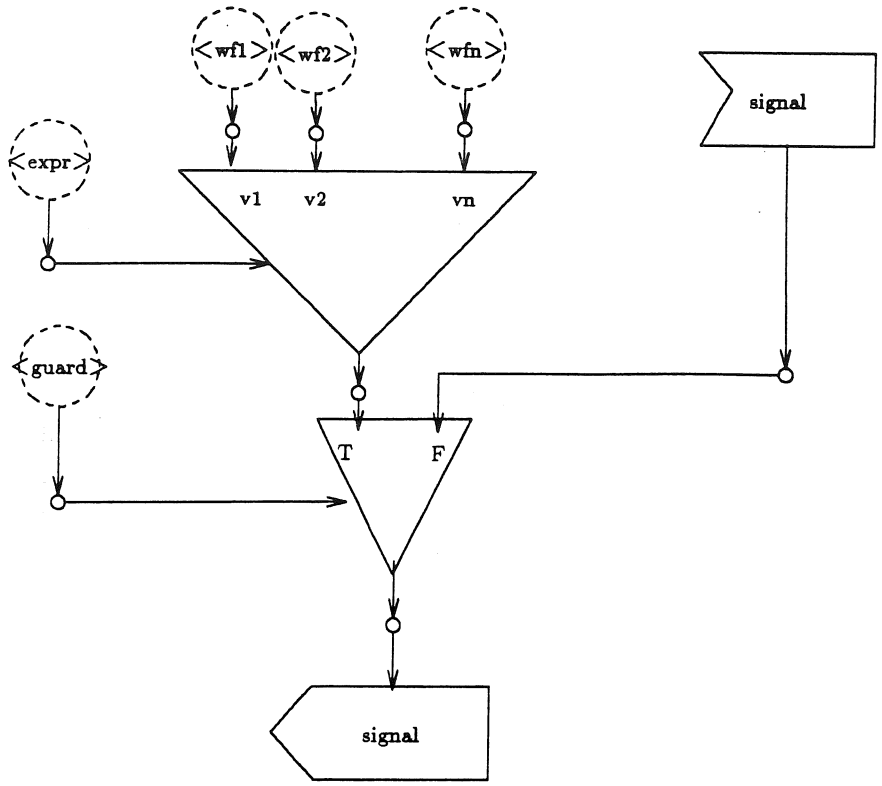


Figure 12: CDFG Choose-Value Node with Guard

the data flow graph of the corresponding condition <cond$_i$>. The **true** input of that choose-value node is connected to the data flow graph of the associated waveform <waveform$_i$ >. Its **false** input is connected to the output of the next condition <cond$_{i+1}$> which is to be evaluated in the case that cond$_i$ is **false**. If the conditional signal assignment is guarded then an additional choose-value is inserted at the bottom of this chain. This assures that the assigned value can only pass if the guard is **true**. The output of the bottom most choose-value node is connected to the **write** node of the signal.

3.4.13 Loops in the Data Flow Graph

We assume a single-assignment data flow graph. Therefore, the graph does not support constructs for the representation of loops. Loops are, however, supported within the control flow graph.

signal <= [**guarded**]
<waveform1> **when** <cond1> **else**
<waveform2> **when** <cond2> **else**
...
<waveformn> **when** <cond$_n$> **else**
<waveform$_{n+1}$ >;

Figure 13: VHDL Conditional Signal Assignment with Guard

3.4.14 Events in the Data Flow Graph

Events at the data flow graph level generally refer to either events on the clock signal or to the specification of asynchronous behavior of components, such as the asynchronous load of a register. Events are represented by explicit **event nodes** in the data flow graph. An event node is a diamond-shaped node labeled by the type of event (Figure 2). An event node can be considered as a special operator node; namely, a predicate that tests for the presence of an event on a signal. It takes a signal as input and produces a boolean value as result.

In VHDL, such events are introduced by signal-related attributes. The following four signal-related attributes are of interest for synthesis and are thus supported by our model: SIGNAL'EVENT, SIGNAL'STABLE(delay), SIGNAL'RISING and SIGNAL'FALLING . The first two are predefined by VHDL. The last two may be defined as follows:

SIGNAL'RISING $\equiv_{DEF}$ SIGNAL'EVENT and (SIGNAL='1')

SIGNAL'FALLING $\equiv_{DEF}$ SIGNAL'EVENT and (SIGNAL='0')

All four attributes accept a signal as argument and return a boolean value as result. The STABLE attribute optionally takes a time expression as second argument. The attribute SIGNAL'STABLE(< $time - expression$ >) returns the value **true** when an event has not occurred on the signal SIGNAL for < $time - expression$ > units of time. Otherwise, it returns the value **false**. The attribute SIGNAL'EVENT returns the value **true** when an event has just occurred on the signal SIGNAL; otherwise, it returns the value **false**. These signal-related attributes may appear anywhere in a VHDL specification where a condition is being specified. In other words, they may be used in the condition field of a conditional statement, in the guard of a block statement, and in a loop test.

An example of how these signal-related attributes may be used in a conditional signal assignment statement is shown in Figure 15. The CDFG representation of this

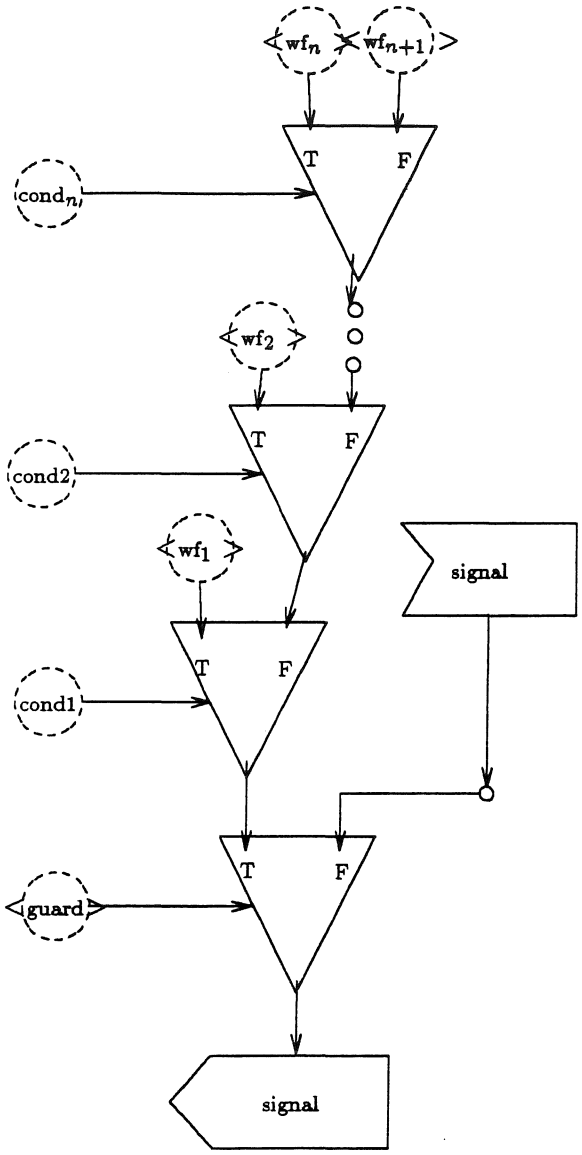


Figure 14: CDFG Representation for a Conditional Signal Assignment

```
OUT <=
   IN1 and IN2  when (IN1'RISING and IN2='1')  else
   IN1 and IN2  when (IN1'FALLING and IN2='0')  else
   IN1 and IN2  when (IN2'RISING and IN1='1')  else
   IN1 and IN2  when (IN2'FALLING and IN1='0');
```

Figure 15: Signal-Related Attributes in a Conditional-Signal Assignment Statement

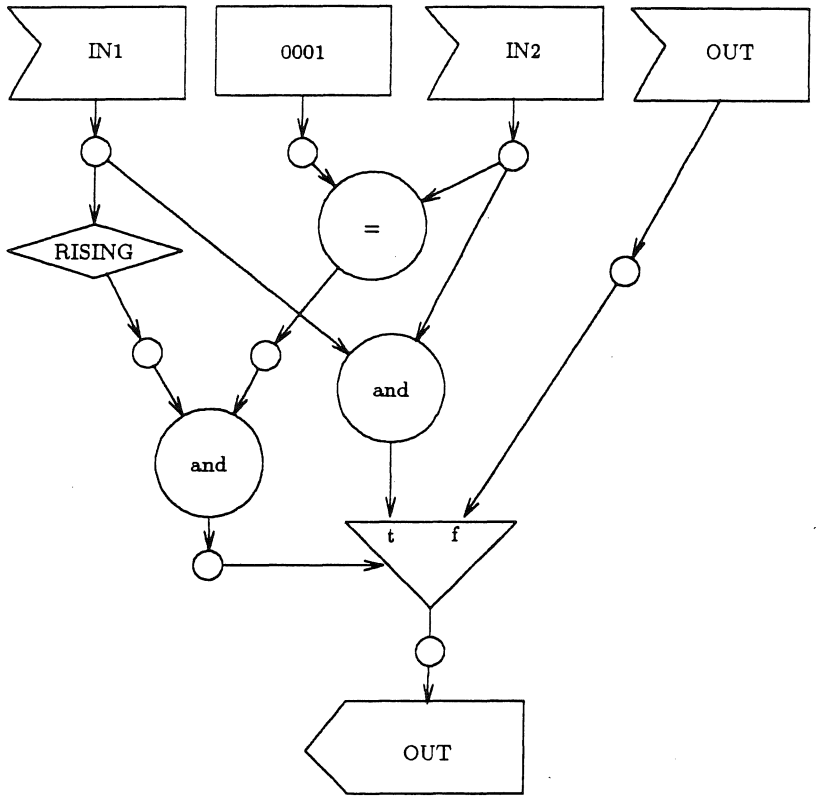


Figure 16: Events in the Data Flow Graph

VHDL specification (Figure 16) depicts the first assignment of the conditional signal assignment statement. The other three assignments can be represented by a similar graph.

3.4.15 Path-Related Timing Constraints

We classify all timing specifications into two groups, namely, the **path-related** and the **event-related** timing constraints. Path-related timing constraints are discussed in this section, while event-related timing constraints are presented in the next section.

A **path-related** timing constraint models the time taken for the effect of a signal to propagate through a set of hardware units. In other words, it indicates a **delay** for the transfer of data from one point of the hardware to another. A **path delay** node, or short **delay** node, can be used to model the delay of an individual operator node as well as the delay of a group of operator nodes. It can also capture point-to-point delays from one particular input of an operator to the output.

In the design representation, we represent such a path delay by the a **path delay** node and a set of associated **timing arcs**. A delay node has one or more input timing arcs (that identify the source nodes) and one output timing arc (which identifies one destination node). The path delay then represents a constraint on the execution of nodes that lie on directed data flow paths between the source nodes and the destination node. The delay node has attributes of two different information types. The first is an ordered list of event specifications and the second is a list of associated timing specifications.

- **event specification:**
  - **sources**: one or more incoming timing arcs from source signals;
  - **destination**: one outgoing timing arc to the destination signal;
  - **event-type**: we distinguish between the three event types RISING, FALLING, and CHANGING; one of these event types is associated with each of the source signals and the destination signal.
- **timing specification:**
  - **delay duration**: minimal, nominal and maximal delay values for the timing constraint value;
  - **delay value**: integer number to indicate the delay in nano seconds.

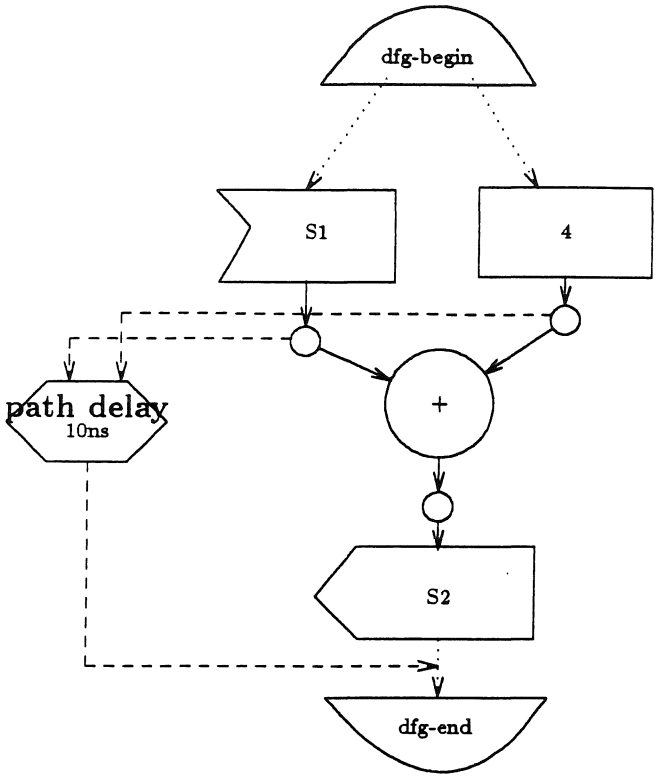
An event specification associates an event-type with each timing input arc (source signals) and with the timing output arc (destination signal). The timing arcs identify a subgraph of the flow graph, to which the delay node is referring to, by pointing to sources signals and to the sink signal. The delay described in the path delay node constrains the data flow nodes on the paths from the source nodes to the destination node.

The event-type corresponds to one of the following three values: RISING, FALLING, or CHANGING. These three event types are represented by the symbols $\uparrow$, $\downarrow$, and $\uparrow\downarrow$, respectively. If no event-type is given for a signal, then the event-type CHANGING is assumed as default. Note that path delay constraints most often refer to delays of data values with n-ary bit-width (n > 1) rather than single-bit control signals, and in this case the event-type CHANGING is the only applicable event specification. We introduce this event specification in order to indicate for what event type the timing specification is given. This is needed since different delays may for instance be specified for when an input signal is rising than for when it is falling.

The timing specification portion of a delay node specifies minimal, nominal, or maximal timing constraint values. A timing specification consists of a timing operator and delay value pair. If no timing duration is given, then the default duration "nominal" is assumed. The timing operator takes the form <, =, and >, which means "at least" (or minimal), "approximately equal" (nominal), and "at most" (maximal). The delay value gives the corresponding time delay in form of a constant value – possibly in multiples of the predefined CYCLE constant.

In VHDL, a path-related timing constraint can be specified by the **after-clause** construct. This construct has the form "**after** <time-expression>". It may be appended to signal assignment statements but not to variable assignment statements. Thus, delays are specified in VHDL relative to the reading or writing of signals. An example of a simple path delay is given in Figure 17. We can make the following observations concerning the delay node attributes previously discussed:

- **event specifications:**
  - (**source** S1, **event-type** change)
  - (**source** node4, **event-type** change)
  - (**destination** S2, **event-type** change)
- **timing specification:**
  - (**delay duration** nominal, **delay value** 10ns)



VHDL description: S2 <= S1 + 4 **after 10ns;**

Figure 17: A Simple Path Delay.

In VHDL, such timing constraints may also be specified in a conditional or a selected signal assignment statement. Each assignment within these statements can be extended by an **after**-clause (Figure 18). In this context, the specified path delays are event- or condition-dependent. In other words, the delay is relative to the occurrence of some event or some condition. For this, the delay node has an additional input port connected to the cause of the delay.

In Figure 18, we present a VHDL description of conditional path delays. Both assignments within the selected signal assignment statement have an associated **after**-clause. The corresponding CDFG representation is given in Figure 19. Since the delays depend on the evaluation of the COND expression, both delay nodes have the node "COND" as one of their input timing arcs.

```
with COND select
S <= A + B  after d1  when C1,
     A - B  after d2  when C2;
```

Figure 18: VHDL Specification of a Path-Delay.

There are two timing delay nodes. The attributes of the first delay node are:

- **event specifications:**
  - (**source** COND, **event-type** change)
  - (**source** A, **event-type** change)
  - (**source** B, **event-type** change)
  - (**destination** S2, **event-type** change)
- **timing specification:**
  - (**delay duration** nominal, **delay value** d1)

The attributes of the second delay node on the right-hand side of Figure 19 are:

- **event specifications:**
  - (**source** COND, **event-type** change)

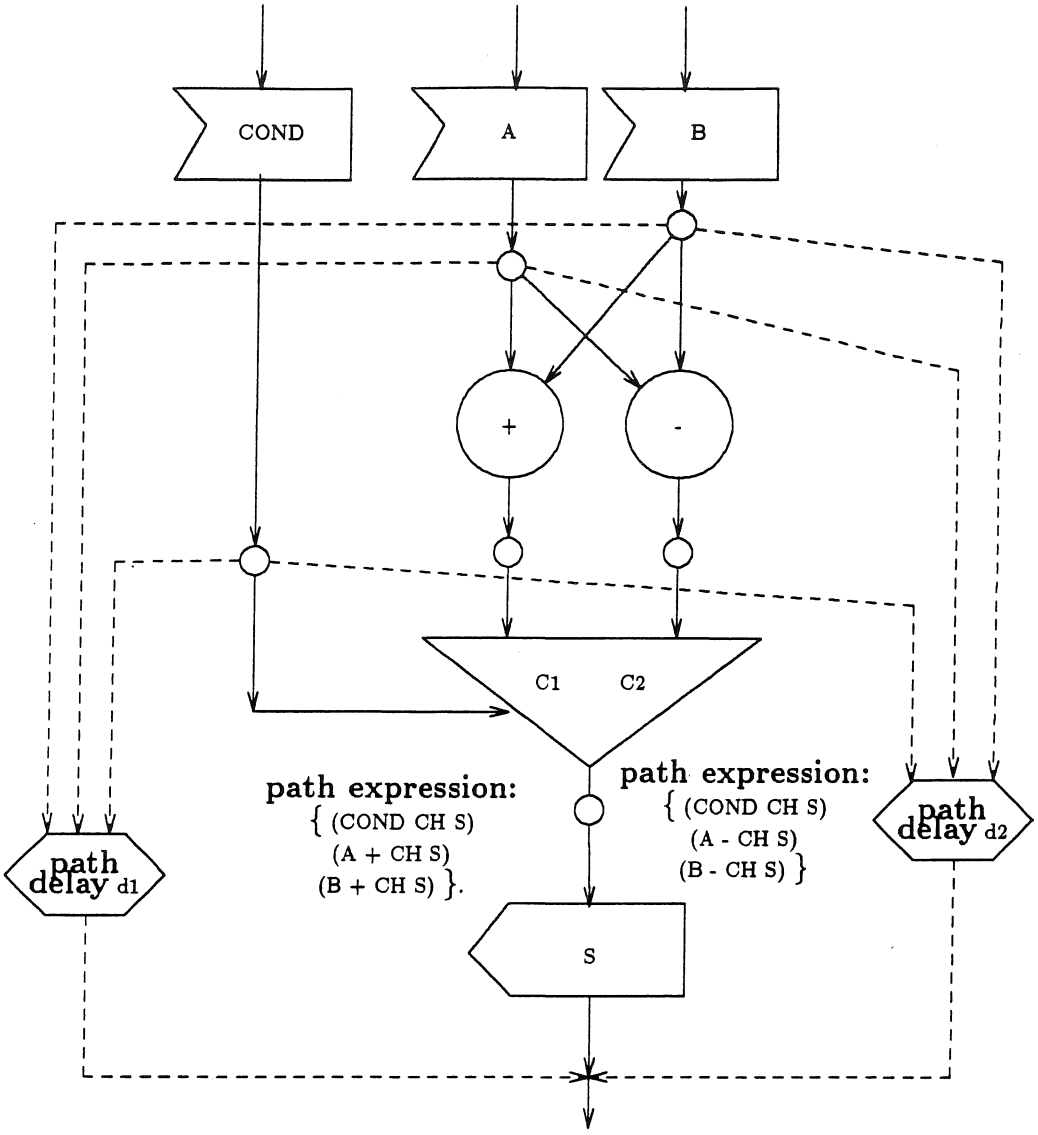


Figure 19: VHDL Specification of Path Delays. (Selected Signal Assignment with After-Clauses.)

- (**source** A, **event-type** change)
- (**source** B, **event-type** change)
- (**destination** S2, **event-type** change)

- **timing specification:**
  - (**delay duration** nominal, **delay value** d2)

The textual example specification (Figure 18) clearly indicates the assignment statement to which each delay refers. For instance, delay d1 refers to the expression containing the Addition operator and delay d2 to the expression with the Subtraction operator. In graph-theoretic terminology, the delays d1 and d2 describe delays of two different paths. However, the design representation depicted in Figure 19 (even in combination with the detailed timing specifications listed in the previous paragraph) is ambiguous. That is, it is not clear which path has a delay of d1 and which has a delay of d2. Reasons for this are that the VHDL expression is compiled into one single data flow graph; and the two paths representing the two expressions are intertwined within this graph. In addition, both timing constraints have the same source and destination nodes.

We can thus conclude that path delays cannot be represented by simple point to point delay arcs as generally suggested in the literature [5]. There may be several directed paths of data flow arcs from the source nodes of a delay node to its destination node. However, a path delay does not always specify a constraint for all paths between the sources and the sink. Instead, the constraint could be restricted to a subset of these paths. We thus need a mechanism to describe the paths of the data flow graph that a delay specification is referring to. We solve this problem by extending the **delay** node by the concept of a **path expression**. A **path expression** specifies a list of one or more paths in the data flow graph. Each path is represented as an ordered list of data flow node identifiers starting with one of the input source nodes and ending with the destination node. A **path expression** is defined more precisely below.

Definition 2 *Let D be a path delay specification in the data flow graph DFG. A* **path expression** *P corresponds to a list of paths,* $P = \{p_1, p_2, ..., p_n\}$. *Each path p in P is of the form* $p = (n_1 n_2 .... n_m)$, *where*

- n_1 *is one of the source nodes of the delay D, and*
- n_m *is the destination node of the delay D, and*

- *for all n_i, n_{i+1} in p, there is a directed data flow arc in the flow graph G from n_i to n_{i+1}.*

For all source nodes m_i of the delay D, there is at least one path p in P with m_i in p.

The design representation for the example VHDL description given in Figure 18 is updated accordingly. In other words, each delay node in Figure 19 is annotated with a path expression. For the delay node on the left-hand side of Figure 19 the expression is:

- **path expression:**
  - { (COND CH S),
  - (A + CH S),
  - (B + CH S) }

The path expression for the delay node on the right-hand side of Figure 18 is:

- **path expression:**
  - { (COND CH S),
  - (A - CH S),
  - (B - CH S) }

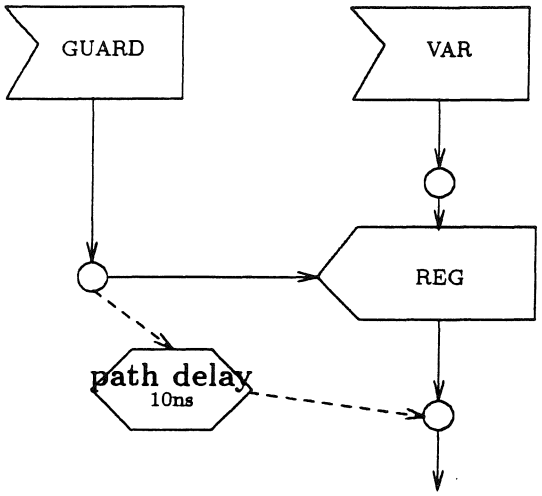
Due to the introduction of the path expressions, the design representation is no longer ambiguous. It is for instance clear now that delay d1 refers to the Addition and delay d2 to the Subtraction operator.

In the VHDL timing specifications we have discussed so far assignment statements are made to signals. Signal assignment statements can however also be used to model sequential circuits, like for instance a register. An example of such a guarded signal assignment statement is given next:

signal REG: INTEGER range 0 to 4095 **register**;

REG <= **guarded** VAR **after** 10ns;

In this case, the specified delay of 10ns is relative to the guard, i.e., the clock, rather than being a delay from the inputs of the register to the output. Thus, the



VHDL description: REG <= **guarded** VAR **after** 10ns;

Figure 20: Modeling the Delay of a Register.

source of the register delay is the CLK and the sink is the value written to the register. Since a sequential circuit is a clocked device, the representation of the delay (Figure 20) is different than the path delay shown in Figure 17.

The presented timing constructs for modeling path delays in the CDFG graph are more powerful than the semantics of the VHDL **after**-clause. The **after**-clause generally refers to a whole group of operations, that is, an expression. It always specifies delays from all inputs of the expression to the output. Point-to-point delays from one single input to one output can on the other hand not easily be specified. Furthermore, a VHDL after-clause always specifies a nominal delay (used by the simulator to schedule updates in the future); it supports no mechanism to give maximal or minimal timing constraints. In the following section we will discuss how VHDL can be extended to handle the specification of these more fine-grained timing constraints.

3.4.16 Event-Related Timing Constraints

Timing constraints that specify delays between the occurrences of events are called **event-related timing constraints.** For example, two signals may be required to have valid values during the same time interval, or a particular data signal needs to be available for reading a certain time interval after the address line signal has gone high. These relationships of events are often expressed by designers via timing diagrams (like the one shown in Figure 21). The event-related timing constraints are thus particularly useful for specifying handshakes and other interface protocols.

For the representation of the event-related constraint type in the data flow graph we use a timing constraint node with the label **event-delay**, or short **event**. The event-related timing constraint node is identical in format to the path delay constraint node. The only exception is the fact that path expressions are not associated with the former construct. Events constrained by an event-related timing constraint need not necessarily be related by data flow arcs, and therefore a path between them may not exist.

To explain the format of the event-related timing constraint, we give an example specification:

DATA1$\uparrow$, DATA2$\downarrow$, DATA3$\uparrow\downarrow$: $>$ 10ns, $=$ 20ns, $<$ 30ns.

This specifies the following timing constraint between the events on the values of DATA1, DATA2, and DATA3: From the time that DATA1 is rising and DATA2 is falling, there will be at least 10 ns, in average 20ns, and at most 30 ns before

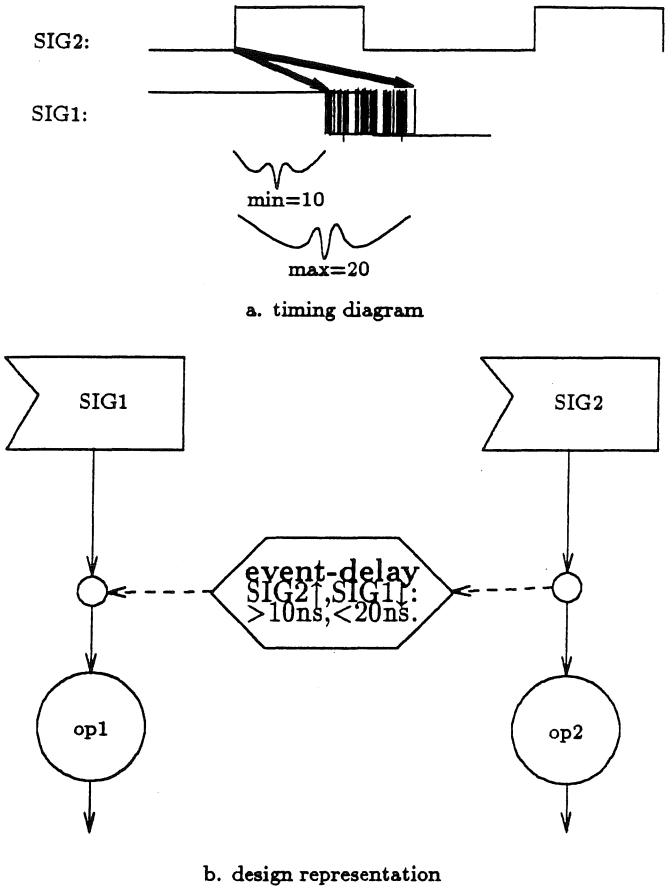


Figure 21: The Representation of Event-related Timing Constraints.

an event occurs on DATA3. The event delay node that captures this example delay specification can thus be described by the following timing attributes:

- **event specifications:**
  - (**source** DATA1, **event-type** rising (↑))
  - (**source** DATA2, **event-type** falling (↓))
  - (**destination** DATA3, **event-type** change (↑↓))
- **timing specifications:**
  - (**delay duration** minimal, **delay value** 10ns)
  - (**delay duration** nominal, **delay value** 20ns)
  - (**delay duration** maximal, **delay value** 30ns)

The proposed timing modeling constructs are powerful and allow the user to create arbitrarily complex signal timing schemes. We can for instance model **duration timing constraints** associated with sequential circuits, such as, setup and hold times. Duration timing constraints refer to the duration during which a signal has to be stable rather than to a delay needed to traverse a path in the underlying structure. The event-related delay constraint can also specify the characteristics of a clock signal by denoting the delay from the rising edge to the falling edge of the clock, or vice versa, and the delay from the falling to the rising edge.

Figure 22 gives a timing diagram for set-up times, hold times, and delay times for a register. Figure 23 then shows how these different timing constraints would be represented in the CDFG graph. Setup and hold times are event-related timing constraints and thus are represented by **event-delay** timing nodes. The propagation delay of the register, however, is a path-related timing constraint and is thus represented by a **path-delay** node.

Setup time is defined as the minimum time that the input should continue to be stable before the clock becomes active. It can thus be modeled by the following event-related timing constraint between the input to the register and the clock:

DATA↑↓, CLK↑: $>$ 10ns.

This states that at least 10ns have to pass without any change on the DATA signal before the CLK signal can be rising. This can be interpreted to mean that the DATA signal has to be held stable for at least 10ns before the value can be written into a register. This example demonstrates that the duration timing constraint can

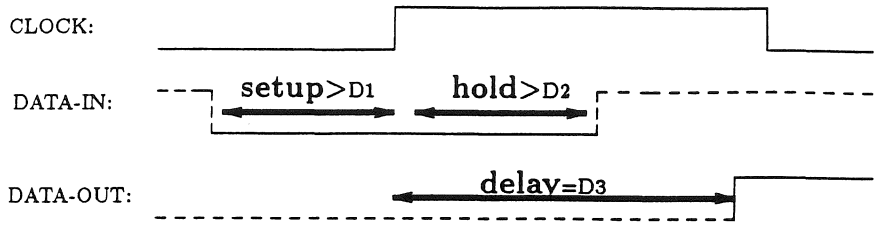


Figure 22: Timing Diagram for Set-up, Hold and Register Delays.

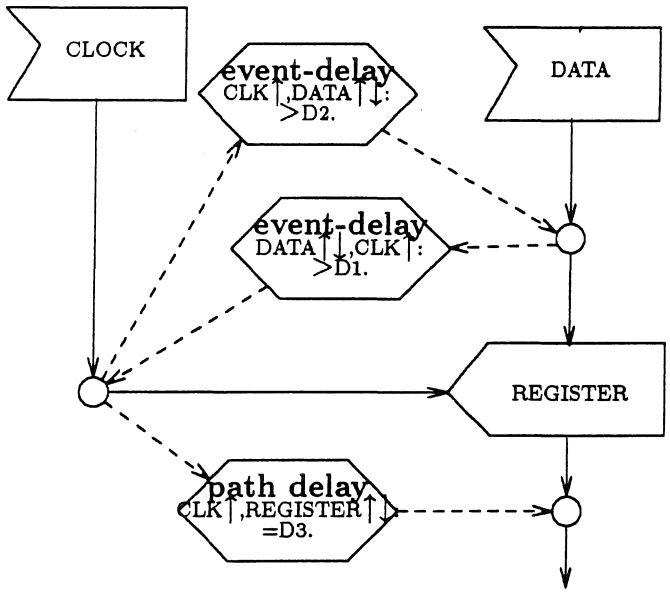


Figure 23: CDFG Representation of Set-up and Hold Times.

be modeled by the event-related timing constraint on the two events of initiating the WRITE operation and on the actual value written. The timing attributes for the setup event delay node can be summarized as follows:

- **event specification:**
  - (**source** DATA, **event-type** change (↑↓))
  - (**destination** CLK, **event-type** rising (↑))
- **timing specification:**
  - (**delay duration** minimal, **delay value** 10ns)

Hold time refers to the minimum time that the input should continue to be stable after the clock has become active. Hold time can thus be specified by the following:

CLK↑, DATA↑↓: > 30ns.

It states that from the time the signal CLK is rising there will be at least 30ns before the DATA value signal may be changed. In other words, the value DATA has to be held stable for at least the hold interval (at least 30ns) after the write operation has been initiated. The timing attributes for the event delay node that models the hold time can be summarized as follows:

- **event specification:**
  - (**source** CLK, **event-type** rising (↑))
  - (**destination** DATA, **event-type** change (↑↓))
- **timing specification:**
  - (**delay duration** minimal, **delay value** 30ns)

Event-related timing relationships cannot easily be specified by VHDL descriptions. The only timing construct of VHDL, the **after-clause**, has to be appended to signal assignment statements. Thus, an **after-clause** is associated with the active modification of signal values which is not appropriate for modeling most event-related delays. Therefore, we need another mechanism for specifying event-related timing constraints in a VHDL textual description. There are three alternatives we have considered, which are to define timing constraints via assertion statements, via special comments, and via a pseudo procedure call.

The expression of assertion statements to represent these timing constraints would get quite clumsy. For instance, the timing specification of

DATA↑↓, CLK↑: > 10ns.

would be expressed by the following assertion statement:

```
assert ((not CLK'stable(0ns) and CLK='1')
        and (DATA'last_active > 10ns))
        or (CLK='0') or (CLK'stable(0ns))
report "setup time violation"
severity "warning";
```

An assertion statement is a sequential VHDL statement. It can therefore only be used within a sequential VHDL description, such as, a process or a subprogram, but not within a concurrent block. Therefore, assertion statements are not a satisfactory means for specifying timing constraints for synthesis.

Instead, we suggest that timing relationships of signals are expressed via a comment statement in a VHDL specification. The syntax of the comment statement for specifying event-related timing constraints is as follows:

- - TIMING: < timing specification >

where the term <timing specification> is an expression of the format described earlier.

It is equally possible to develop a VHDL procedure for specifying the timing constraints. Below, we give an example of such a special-purpose procedure:

```
procedure TIMING-CONSTRAINT (<timing specification>) is
begin
# empty body here
end TIMING-CONSTRAINT;
```

All information specified in the previously discussed timing specification is entered into this procedure in the form of parameters. The body of this pseudo procedure is empty, and thus the simulation of the VHDL specification would not be affected by a call to this procedure. The compiler however would interpret a call to this pseudo-procedure as a hint (a comment) to insert timing constraints into the design representation for synthesis. A procedure specification for simple one-source to one-sink delays is introduced in [22].

4 THE CONTROL FLOW GRAPH

In this section, we discussed the control flow graph object types of the augmented CDFG model. A control flow graph is created for loops, conditional statements, and other control constructs of the specification, such as a procedure, a subprogram or a process.

4.1 The Control Flow Graph Definition

Below, we first define the graph and then discuss its object types in more detail.

Definition 3 *A* **control flow graph** *is a directed (not necessarily acyclic) graph CFG = (CN,CE,CP,CF) with CN the set of vertices, CE the set of edges, CP the set of ports, and CF the set of control flow marking functions. The elements of CN are uniquely identified by the function* **vertex-num**: $CN \rightarrow INTEGER$.

1. *CN corresponds to the set of control flow nodes. It is composed of several disjoint sets,* $DN = PROCESS \cup CALL \cup CONDITION \cup STMT\text{-}BLOCK \cup MARKER \cup TIME$.
   - *PROCESS is the set of process nodes.*
   - *CALL corresponds to the set of procedure call nodes.*
   - *STMT-BLOCK corresponds to the set of statement blocks.*
   - *CONDITION is the set of condition and event nodes.*
   - *MARKER corresponds to the set of demarcation nodes.*
   - *TIME corresponds to the set of timing constraints.*
2. *CP is the set of ports. They corresponds to the connection points of vertices with the arcs of the graph. The function* **port-class**: $P \rightarrow \{input\text{-}port, output\text{-}port\}$ *specifies whether a port is an input or an output port. A vertex* $v \in CN$ *can have an ordered (possibly empty) list of input and output ports* $p \in CP$, *respectively. The function* **input**: $CN \times INTEGER \rightarrow CP \cup \emptyset$ *is an assignment of input ports to vertices. The function* **output**: $CN \times INTEGER \rightarrow CP \cup \emptyset$ *is an assignment of output ports to vertices.*

3. *CE represents the set of directed edges between the (ports of the) vertices of the graph. The edges correspond to pairs (p1,p2) $\in$ CP $\times$ CP with the direction of the edge from p1 to p2. We distinguish between five types of edges, which are control flow arcs, concurrent arcs, hierarchy arcs, timing arcs, and demarcation arcs.*

4. *CF is a set of control flow marking functions CF_i: CN $\rightarrow$ CM with CM the set of possible attribute domains.*

4.2 Representation of the Control Flow Vertices

The graphical depictions shown in Figures 24 and 25 are used to represent the constructs of a control flow graph. The meaning and attributes of each vertex type is explained below.

Type: Concurrent nodes

Graphic: Parallelogram with the labels called **co-begin** and **co-end**

Description: A **co-begin/co-end** node pair marks a concurrent piece of code (in this report we referred to as a block). All control nodes attached to the **co-begin** node by outgoing edges execute simultaneously (in different parts of the same data path or even in different processors). Thus, a **co-begin** node forks possibly independent processes. A **co-end** node simply collects these concurrent processes into one location.

A **co-begin** node has one input and numerous output edges (of the concurrent edge type). A **co-end** node has one output and numerous input edges (of the concurrent edge type). The outgoing arcs of a **co-begin** node and the incoming arcs to a **co-end** node demark the concurrent pieces of code that are contained within the concurrent specification modeled by the **co-begin/co-end** node pair. A **co-begin/co-end** node pair can model hierarchically nested concurrent blocks, since the concurrent block modeled by a **co-begin/co-end** node pair could contain other concurrent blocks. The incoming arc of a **co-begin** node originates at a control flow node representing the block in which the concurrent piece is contained. Similarly, the outgoing arc of a **co-end** node points back to the control flow node that represents the end of a block in which the concurrent piece is contained.

A **co-begin** node has one or more outgoing edges to control flow nodes out of which the concurrent block is composed of. They point to processes, blocks and concurrent procedure calls. One of these outgoing arcs may point to a statement-block node. The data flow graph associated with the latter contains all concurrent signal assignments specified in the block description. A concurrent block can also

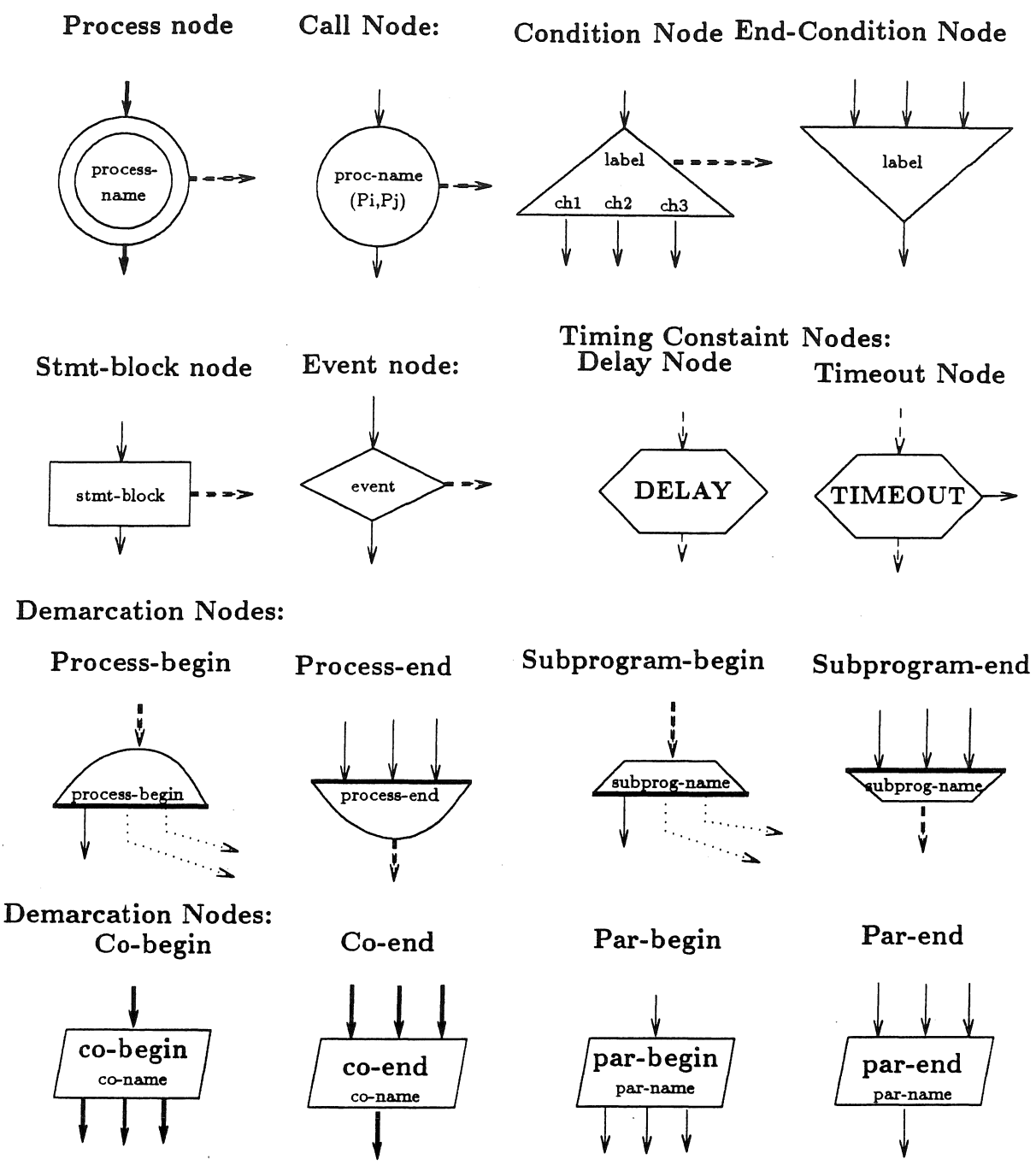


Figure 24: Graphical Representation of Nodes in the Control Flow Graph

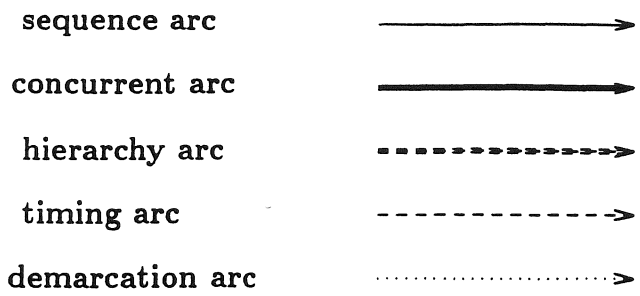


Figure 25: Graphical Representation of Arcs in the Control Flow Graph

have a guard, i.e., a condition which determines under which event the block is to be executed. In this case, a data flow expression graph that represents this guard is included in the data flow graph associated with the statement-block node of the concurrent block.

Type: Process node

Graphic: Double circle with process name

Description: A process node represents a process specification. A process node has one concurrent input arc and one concurrent output arc to the predecessor and successor nodes in the control flow graph, respectively. It also has one incoming and outgoing hierarchy arc to its process specification, i.e., to a **process-begin** and **process-end** node pair. A process node has an (optional) sensitivity list associated with it.

Type: Statement-block node

Graphic: Rectangle

Description: A statement-block node represents a number of assignment and data selection statements that are represented by a data flow graph. It has one input and one output control flow arc to its predecessor and its successor in the control flow graph, respectively. It also has one outgoing hierarchy arc that points to the representation of the associated data flow graph.

Type: Call node

Graphic: Circle with the procedure name

Description: A procedure call is represented by a call node in the control flow graph. A call node has one input control flow arc from its control flow predecessor. It has one output control flow arc to indicate where to continue in the control flow sequence once the associated procedure has been executed. There is one additional outgoing arc of type hierarchy to the corresponding procedure specification.

The representation of parameter passing is handled as follows. If a parameter of mode **in** corresponds to an expression rather than a variable name, then a data flow graph that represents this expression and assigns the result to a temporary variable is generated. This data flow graph is placed into the control flow node prior to the procedure call. A possibly empty list of read/write and constant nodes corresponding to the actual parameters (or the temporary variables that hold the value of the actual parameters) is associated with the call node. This list is ordered by position, i.e., the first actual parameter is passed to the first formal parameter, and so on.

Type: Condition node

Graphic: Triangle with a condition label

Description: A **condition** node distributes control among a number of control flow sequences. This node type is used to model loop statements and conditional statements. The attribute function **condition-type**: CN $\rightarrow$ { IF, CASE, FOR, WHILE, REPEAT } assigns a condition label to a **condition** node. A **condition** node has one input control flow arc from the predecessor in the control flow graph, and two or more output control flow arcs to mutual exclusive successors in the graph.

If the represented condition corresponds to a simple boolean or integer variable, then the **condition** node is annotated by the name of that variable. If the condition corresponds to an expression, then a data flow graph that represents this expression and assigns the result to a temporary variable, called the conditional variable, is generated. It is attached to the **condition** node. Each output port of the condition node has a constant guard against which the conditional variable is compared. The output branch associated with a guard will be executed when the value of the conditional variable equals the value of that guard.

Type: End-condition node

Graphic: Reversed triangle with a condition label

Description: An end-condition node is used to mark the end of a conditional statement. It collects all threads of computation that arise from the corresponding **condition** node. Thus, an end-condition node has one input port for each output port of the matching **condition** node. The attribute function **condition-type:** CN → { END-IF, END-CASE } assigns a label to an **end-condition** node.

Type: Parallel nodes

Graphic: Parallelogram with the labels **par-begin** and **par-end**

Description: There are two nodes to demark parallel threads of control, called **par-begin** and **par-end.** A **par-begin** node has one input and two or more output control flow edges. A **par-end** node has one output and two or more input control flow edges. At a **par-begin** node, several independent threads are spawned off at the same time. All threads of control execute simultaneous. Control does not proceed past the corresponding **par-end** node, until all threads of controls have finished execution.

Type: Delay node

Graphic: Stop sign symbol with label **delay**

Description: A delay node is used to model a delay timing constraint at the control flow graph level. A delay constraint node specifies a delay for a set of computations. The respective set of computations is marked of by timing arcs associated with this timing node. The attributes **minimal, maximal** and **nominal** specify the minimal, maximal and nominal time or cycle delay, respectively.

Type: Timeout node

Graphic: Stop sign symbol with label **timeout**

Description: A timeout constraint node is used to model a timeout for a set of computations at the control flow graph level. The respective set of computations is marked of by timing arcs associated with the timeout node. There is an incoming timing arc from the source of the constraint and an outgoing timing arc to the sink of the constraint. In addition, a timeout node has an outgoing sequence arc that points to some other node in the control flow graph. A timeout node also has one integer time value attribute, called **timeout value**. When the set of computations associated with the timeout node is entered, then a timer is started up with the **timeout value.** If the constrained set of computations is executed in a time less than or equal to this **timeout value** (i.e., before the timer runs out), then the timeout constraint is met. Then the execution simply continues with the next control flow node in the graph. If the execution of the constrained computations is not completed after the timer has counted up to the **timeout value**, then the computations are interrupted. Execution

then continues with the control flow graph portion that the sequence arc attached to the timeout node is pointing to.

Type: Event node

Graphic: Diamond

Description: An event node in the control flow graph describes an asynchronous event. An event node waits for a particular event or data condition, such as, whether a signal is rising, falling, stable, or whether any event occurred on a signal, before the control execution continues. Input to the event node is a control flow arc from the previous control flow node and output is a control flow arc to the next control flow node that is to be executed after the event has become true. An event node has a hierarchy arc to an associated data flow graph that describes the event that is to be tested for.

Type: Process demarcation nodes

Graphic: Half-circles with double horizontal lines

Description: The demarcation node pair **process-begin** and **process-end** mark off the beginning and the end of a process specification. The **process-begin** nodes have one input hierarchy arc from the corresponding process node that is being represented. The **process-end** nodes have one output hierarchy arc that points to the corresponding process node. A **process-begin** node has zero or more output demarcation arcs that point to **subprogram-begin** nodes of subprograms that are defined or used within that process. It also has a control flow arc to the first control flow node within its process body.

Type: Subprogram demarcation nodes

Graphic: Half-sixatons

Description: The demarcation node pair **subprogram-begin** and **subprogram-end** mark off the beginning and the end of a subprogram representation, respectively. The **subprogram-begin** demarcation node has zero or more arcs that point to other **subprogram-begin** nodes that are defined or used within that subprogram. It also has one output control flow arc to the first control flow node in the control flow graph that represents the behavior of the subprogram. The **subprogram-end** node collects all control flow arcs that exit the subprogram. The node pair is connected to its call nodes by hierarchy arcs.

A list of formal parameters and their mode is attached to the **subprogram-begin** node. If the subprogram is a function, then a write node to a temporary

variable named "return-<function-name>" representing the return value is attached to the **subprogram-begin** node as well.

4.3 Representation of the Control Flow Arcs

The graphical depictions of the arc types supported by the control flow graph model are shown in Figure 25. The meaning and attributes of each arc type is explained below.

Type: Control flow arc

Graphic: Arrow

Description: Control flow arcs connect pairs of control flow nodes to show their sequencing. They thus preserve the ordering within the control flow graph.

Type: Hierarchy arc

Graphic: Bold dashed arrow

Description: Hierarchy arcs are used to demark levels of hierarchy within the control flow graph. They connect a control flow node with another control flow graph that describes the behavior encapsulated by the former. For instance, they are used to point from a procedure call node to the body of the procedure. They also point from a **stmt-block** node to the associated data flow graph. In addition, they connect a process node to its **process-begin** and **process-end** node pair.

Type: Concurrency arc

Graphic: Bold arrow

Description: A concurrency arc connects a control flow node to a set of other control flow nodes that are to be concurrently executed with respect to each other. The control flow node at the source of the arc corresponds to a **co-begin** node. Conversely, concurrency arcs are used to connect concurrent control flow nodes to a **co-end** node to mark the end of the concurrent section.

Type: Timing arc

Graphic: Dashed arrow

Description: Timing arcs connect a timing constraint node with other control flow nodes. they denote the portion of the control flow graph that is being constrained by the associated timing node.

Type: Demarcation arc

Graphic: Dashed arrow

Description: Demarcation arcs are used to point from or to demarcation nodes. For instance, demarcation arcs point from a **process-begin** node to all the subprograms specified within the process.

4.4 Modeling with Control Flow Constructs

Below, we show how the just presented control flow model can be used to represent typical specifications written in VHDL.

4.4.1 Process Statement

```
[process-name:] process [( <sensitivity list> )]
    declarative part;
begin
    sequential-statement-part;
end [process-name];
```

Figure 26: A VHDL Process Specification

A VHDL process statement defines an independent sequential piece of code that represents the behavior of some portion of a design. The syntax for a VHDL process statement is given in Figure 26.

In the CDFG model, a process is encapsulated by a **process-begin** and **process-end** node pair. A sensitivity list corresponds to an implicit wait statement; the representation of this is discussed in Section 4.4.14. A process body is composed of a collection of sequential statements. The representation of the process body will thus be discussed in the following sections, since we will show how each of the sequential VHDL constructs is represented.

4.4.2 Subprogram Specification

There are two types of subprograms, namely, procedures and functions. A subprogram specification encapsulates some sequential piece of code into a separate module that can be reused multiple times. The syntax for a VHDL procedure declaration, which only defines the interface of the procedure but not its body is:

procedure < *procedure − name* > (<formal parameter list>);

This corresponds to a forward declaration commonly found in conventional programming languages. The syntax for a complete VHDL procedure specification is given in Figure 27.

---

procedure < *procedure − name* > (<formal parameter list>) **is**
  declarative part;
begin
  sequential-statement-part;
end [< *procedure − name* >];

Figure 27: A VHDL Procedure Specification

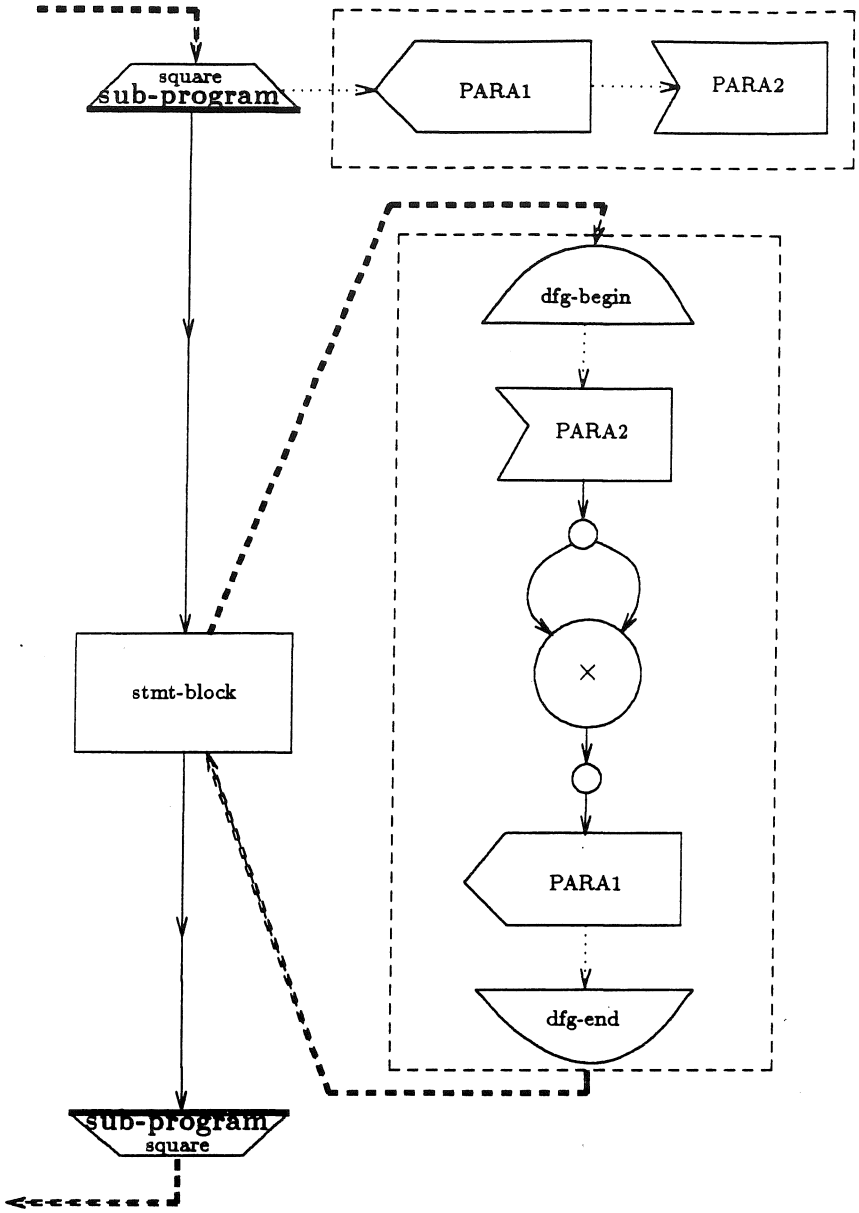
A procedure is represented by a CDFG graph that is encapsulated by a **subprogram-begin** and **subprogram-end** node pair. Figure 28 gives an example of a VHDL procedure specification and its associated CDFG graph.

In Figure 28, a list of read, write and constant nodes is associated with the **subprogram-begin** node. They correspond to the formal parameters of the procedure. This is equivalent to a symbol table for keeping track of the parameters and their modes in the flow graph.

A function specification also encapsulates some sequential piece of code into a separate module. It has no side effects, and returns one value as its result. The syntax for a VHDL function specification is:

function <function-name> (<parameter list>) **return** <type-mark>;

The type-mark corresponds to the subtype of the returned value. Therefore, the list of formal parameters (Figure 28) is extended by one more parameter. This is a temporary variable that holds the return value of the function.



VHDL description:

procedure square (PARA1: **out** Integer, PARA2: **in** Integer) **is**
begin
    PARA1 = PARA2 × PARA2;
end square;

Figure 28: CDFG Representation of the Procedure Specification Statement

4.4.3 Concurrent Block Statement

A VHDL block statement is the primary construct used to represent concurrent descriptions. A block statement groups together a set of concurrent statements that relate to the same portion of the design. The syntax for a block statement is given in Figure 29.

[< *block − name* >:] **block** [(<guard-expression>)]
declarative part;
begin
concurrent-statement-part;
end block [< *block − name* >];

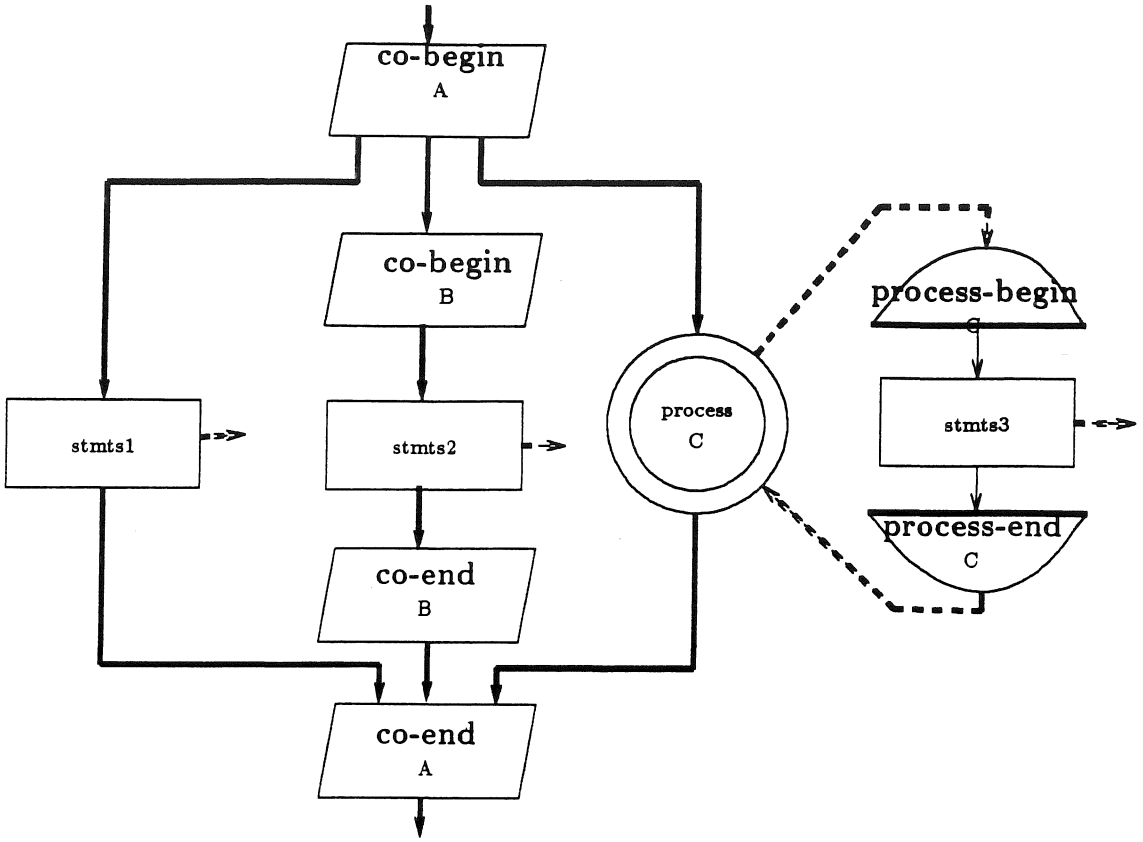
Figure 29: A VHDL Block Specification

A block statement is represented by a **co-begin** and **co-end** node pair. All control flow nodes in between a **co-begin** and **co-end** node pair are assumed to be concurrent.

The block body consists of one or more concurrent statements. There is one statement-block node within each **co-begin** and **co-end** node pair that represents the block body. A data flow graph that represents the concurrent signal assignments statements of a block is associated with the statement-block node. A VHDL block can optionally have a guard, which is a condition which determines under which event the block is to be executed. This optional guard expression defines an implicit signal GUARD. If the optional guard expression evaluates to **true**, all guarded signal assignment statements will be executed, otherwise they are not executed. A data flow graph that represents this guard is included in the statement-block node.

VHDL blocks may be hierarchically nested to support design decomposition. This hierarchical nesting of blocks is shown by the **co-begin** and **co-end** node pairs. There are zero or more outgoing edges from the **co-begin** node to other control flow nodes out of which the block is composed of. Some point to the processes included in the block (**process** nodes), some point to nested blocks (**co-begin** nodes), and others point to concurrent procedure calls (**call** nodes).

An example of a block hierarchy is shown in Figure 30. In Figure 30, the **co-begin** node of block A points to a statement-block node that represents the data-flow graph of all its statements (<statements1>). The **co-begin** node of block B points



VHDL description:

```
A: block
 begin
   statements1;
   B: block
    begin
     statements2;
    end block B;
   C: process
    begin
     statements3;
    end C;
end block A;
```

Figure 30: A VHDL Block Specification

to its statement-block node (<statements2>). Furthermore, a block may contain concurrent procedure calls and process statements. For instance, the outermost block A contains the block statement B and the process C. Therefore, the **co-begin** node of block A points both to the **co-begin** node for block B and a **process** node for process C.

4.4.4 Concurrency versus Parallelism

The **co-begin** and **co-end** node pair is used to model decomposition into concurrent processes, whereas the **par-begin** and **par-end** node pair models parallel threads of execution within a sequentially executing process. The latter concept is commonly expressed by conventional programming languages, such as ADA. There is no explicit construct in VHDL of this kind. VHDL concentrates on the concurrency concept. Concurrency corresponds more closely to the model of hardware units.

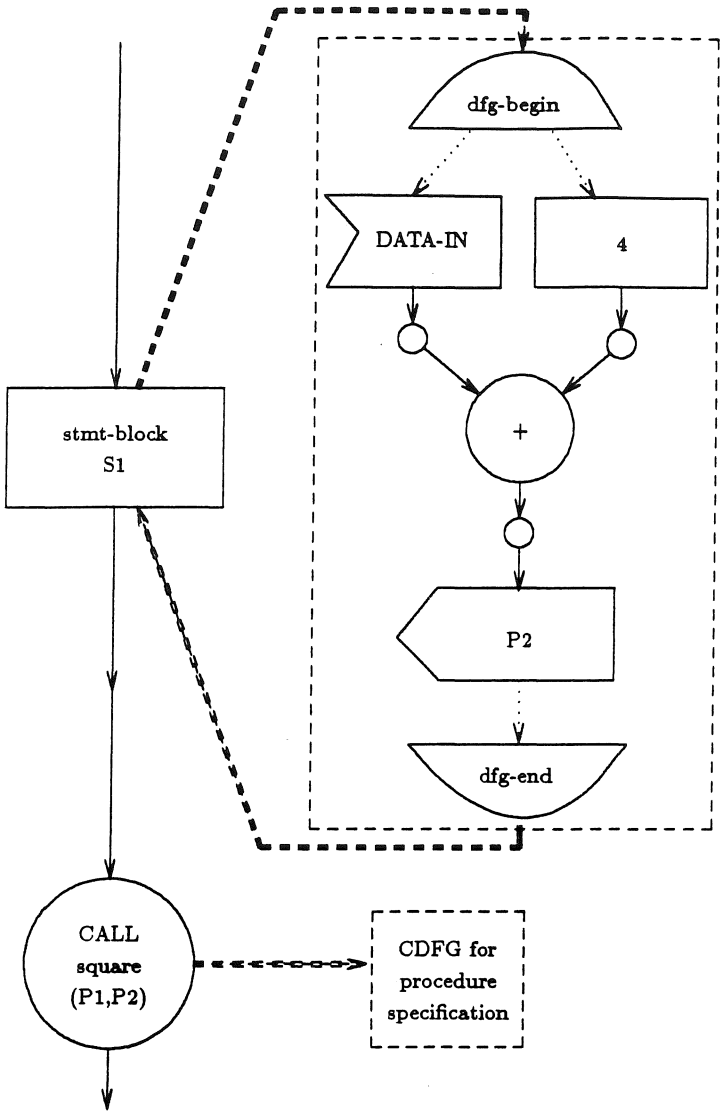
4.4.5 Procedure Call

A procedure call statement is used to invoke a procedure body consisting of sequential statements. A procedure call statement has the following syntax in VHDL:

$< procedure - name > (< parameter - list >);$

As shown in Figure 31, this statement is represented by a procedure call node in the control flow graph. Its outgoing hierarchy arc points to the procedure specification of the called procedure.

The design representation of the procedure parameters is explained by an example. In Figure 31, the procedure call node points via a hierarchy arc to the procedure body that is depicted in Figure 28. A list of actual parameters is associated with the call node. This list contains actual parameter references and temporary variables that hold the value of the actual parameter. The first actual parameter P1 corresponds to the formal parameter PARA1. The second actual parameter corresponds to the second formal parameter PARA2. It is an expression, DATA-IN + 4, rather than a variable name. A data flow graph that represents this expression and assigns the result to a temporary variable, called P2, is generated. This data flow graph is placed into the control flow node prior to the procedure call (Figure 31). The actual second parameter listed in the call node is the temporary variable P2 rather than the complete expression "DATA-IN + 4".



VHDL description: square(P1, (DATA-IN + 4));

Figure 31: CDFG Representation of the Procedure Call Statement

A procedure call statement may be processed by a synthesis system in the following two ways:

1. In-line expansion of a procedure call can be performed. A copy of the CDFG that represents the procedure body is substituted into the current CDFG in place of the procedure call node. In the CDFG copy of the procedure body all occurrences of the formal parameters are replaced by the actual parameters specified in the call node. When this description is synthesized, the hardware that implements the overall CDFG is also used for this section of the CDFG that represents the procedure body.

2. The procedure body can be kept as separate independent entity rather than being in-line expanded. Then, separate hardware is synthesized for the procedure body. Each procedure call provides the values of the actual parameters as inputs to and accepts the values of the modified actual parameters as outputs from the hardware that represents the procedure. This may be done through shared memory or some other means of synchronization (e.g., a stack).

Procedure call nodes encapsulate some arbitrary user-defined behavior by one vertex and thus create *hierarchy* in a control flow graph.

4.4.6 The If-Statement

The VHDL behavioral style has two conditional control constructs, which are the **if**-statement and the **case**-statement. In the CDFG model, control flow nodes are created for each of the behavioral control constructs using pairs of the **condition** and **end-condition** node types (Figure 24). These nodes may be nested and interconnected to directly model the flow of control of the description. In this section, we describe the design representation for the **if**-statement, while the **case**-statement is described in the next section.

The syntax for a VHDL if-statement is shown in Figure 32. An if-statement selects for execution one or none of the enclosed sequences of statements depending on the value of the corresponding conditions. The condition specified in the **if** and **elsif** clauses are evaluated in succession until one evaluates to **true** or all yield **false**. If one condition evaluates to **true** or the **else** clause is found then the sequence of statements associated with this condition is executed.

In the CDFG model, the if-statement is mapped into one or more pairs of **condition** and **end-condition** nodes by the following scheme:

```
if condition1 then
    sequence-of-statements1
elsif condition2 then
    sequence-of-statements2
...
else
    sequence-of-statements_{N+1}
end if;
```

Figure 32: VHDL If-Assignment

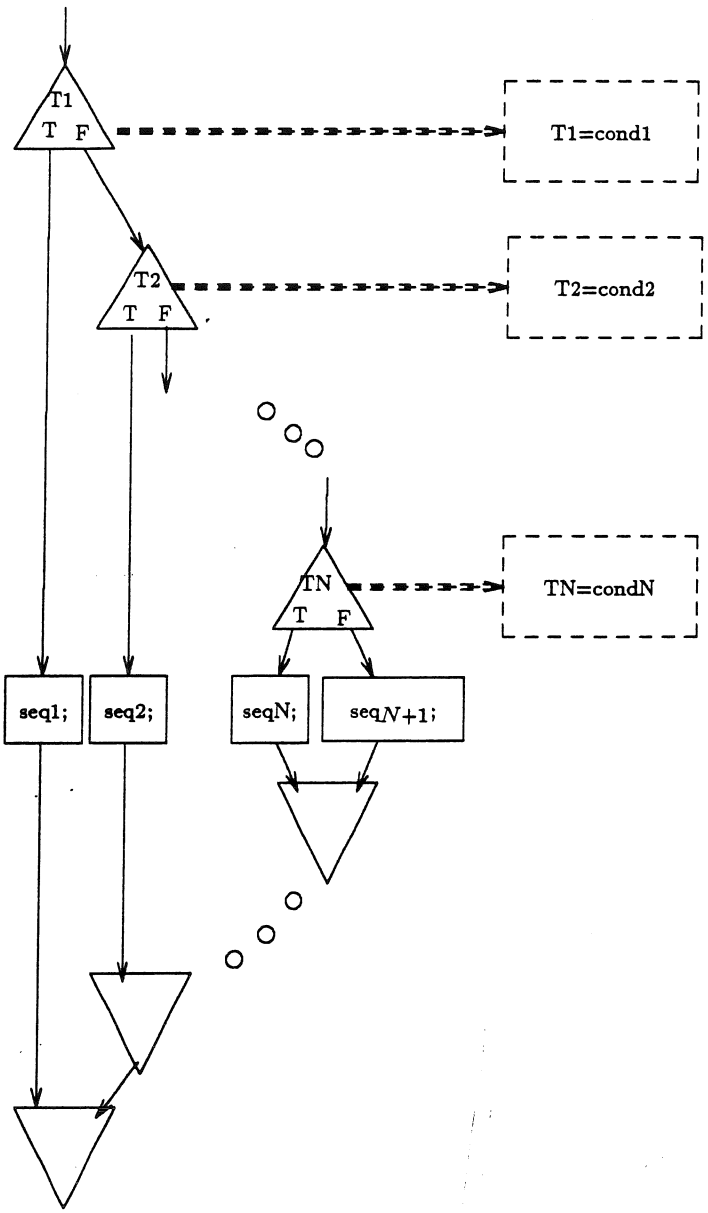


Figure 33: CDFG Representation of the If-Statement

1. A **condition** node is created for each condition following the **if** or the **elsif** clause. With this **condition** node, we associate the label **if** as well as the name of the temporary variable that holds the result of the conditional expression. The data flow graph associated with the **condition** node evaluates the conditional expression and assigns the result to a temporary variable. A **condition** node created for an if-statement has always two outgoing branches, one with the condition value **true** and the second with the condition value **false**. The **condition** node selects the control branch to follow based on the result of the conditional expression.

2. For the **true** branch of each **condition** node, a statement-block node that represents the sequence of statements to be performed in that branch is created[3].

3. The **false** branch of each **condition** node is connected to the **condition** node of the next condition.

4. If there is an **else** clause then the **false** branch of the last **condition** node is connected to its sequence of statements <sequence-of-statements$_{N+1}$>. Otherwise, it is directly connected to the corresponding **end-condition** node.

5. A **end-condition** node is created for each **condition** node. This **end-condition** node collects the corresponding conditional branches.

For an if-statement without the explicit **else** part, the **false** branch of the last **condition** node is directly connected to the corresponding **end-condition** node.

The design representation shown in Figure 33 assumes that the control unit evaluates the condition and then initiates the conditional branching within the same state. Optimizations according to different design styles are possible. First, each test condition could be evaluated in a statement block prior to the corresponding **condition** node. This implies that the condition is evaluated in the state prior to the conditional branching and the result is then stored in a flip-flop (often called a status register). Secondly, it is possible to evaluate all test conditions of a conditional statement prior to execution of the statement[4]. This is so because the result of these different conditional evaluations will not be modified by a partial execution of the statement. In Figure 33, this optimizer would collect all conditions of the form "T_i := $cond_i$" into

[3]If the sequence of statements in the **true** branch contains other control constructs besides assignment statements, then a control flow subgraph that represents these control constructs replaces the statement-block node.

[4]This prior evaluation of all test conditions applies to a single conditional statement only. If there is another conditional statement nested within the first, then the test conditions of the later cannot necessarily be evaluated before the complete nested statement. The reason is that nested conditions may involve values computed somewhere in the computations prior to the branching.

one data flow graph. This data flow graph is then associated with the statement block before the first **condition** node. The advantage of this scheme is that all conditional expressions could be evaluated in parallel. It may result in unnecessary evaluations in the case that one of the early conditions in the if-statement evaluates to **true.**

4.4.7 The Case Statement

The syntax for a VHDL case statement is shown in Figure 34. A case statement selects for execution one of a number of alternative sequences of statements. The choice values <choice$_i$ > and the expression <expr> must be of the same discrete type. Each possible choice value must appear exactly within the case statement. A choice value can be a constant value or a discrete range. The choice value **others** is allowed for the last alternative. It stands for all values not given in the choices of the previous alternatives.

A case statement is represented by a single **condition** node in the CDFG model as shown in Figure 35. The **condition** node has an associated data flow graph that represents the conditional expression <expr>. For each choice value of the **condition** node there is a branch leading to a statement-block node that represents the sequence of statements to be performed in that branch. The **condition** node selects the control branch to follow based on the result of the comparison of the choice values with the conditional expression. An **false** branch of the **condition** node is connected to the statement-block of the **others** branch if it exists. An **end-condition** node collects all branches from the **condition** node.

4.4.8 The For Loop

There are three loop constructs in the behavioral VHDL style: the **for** loop, the **while** loop, and the **infinite** loop. In the CDFG model, loops are generally represented in control flow rather than data flow. They can only be represented in data flow when they are completely unrolled. Each loop construct is modeled by a **condition** node with a condition label that describes the particular condition type. Below, we describe the design representation for the for-loop. The while loop and the infinite loop are discussed in a later section.

The VHDL **for-loop** statement has a loop body that is to be executed repeatedly, zero or more times (see Figure 36). It is controlled by an index variable and a range. The discrete range is first evaluated. If is a null range, then the execution of the loop is complete; otherwise, the index variable's value steps through the specified range

```
case <expr> is
  when <choice> 1 ⇒
   sequence-of-statements-1
 ...
  when <choice_N > ⇒
   sequence-of-statements-N
  when others ⇒
   sequence-of-statements-N+1
end case;
```

Figure 34: VHDL Case Assignment

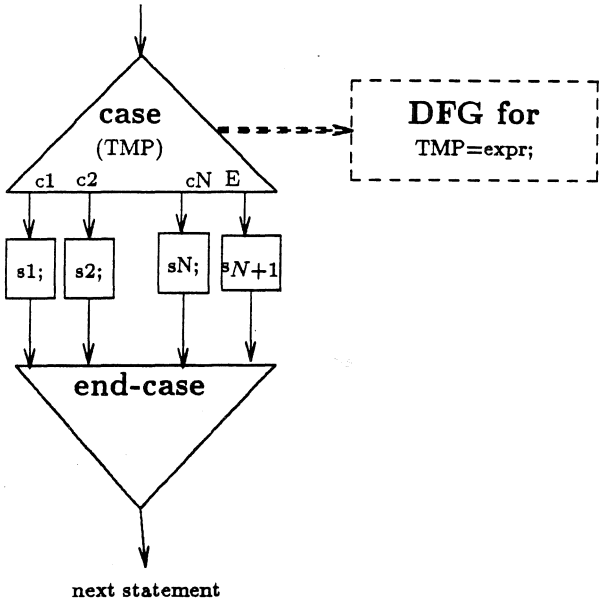


Figure 35: CDFG Representation of the Case Statement

for *identifier* **in** *low* **to** *high* **loop**
sequence-of-statements
end loop;

Figure 36: VHDL For-Loop

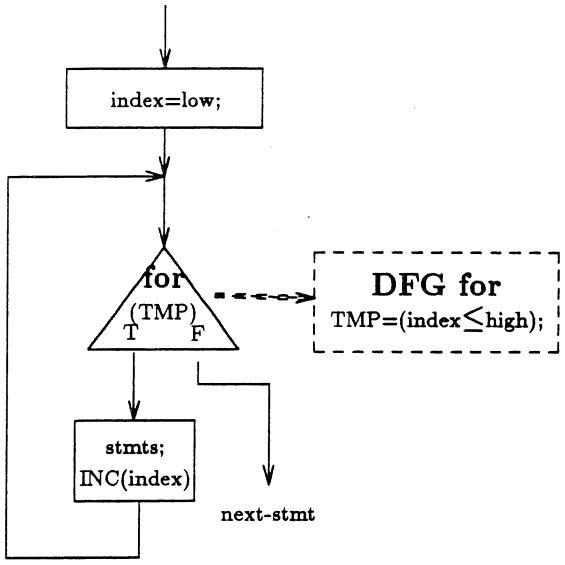


Figure 37: CDFG Representation of the For-Loop

for each iteration of the loop. Prior to each iteration, the current value of the discrete range is assigned to the index variable.

As shown in Figure 37, the for-loop is represented by one **condition** node in the CDFG model. The **condition** node determines whether the loop is to be executed one more time by taking the **true** branch or whether its execution is complete. The **false** branch of the **condition** node is connected to the control flow node that represents the next statement after the loop. At the end of the statement-block node that represents the loop body, the index variable is incremented. Before entering the next iteration, it is tested whether the index variable has reached the upper bound of the range by setting the test bit X := (index $\leq$ high).

4.4.9 The While Loop

The VHDL **while-loop** construct is shown in Figure 38. The condition of a **while-loop** is evaluated before each execution of the sequence of statements. If the value of the condition is **true** then the statements are executed. If it is **false** then the loop is complete.

The while loop is represented by one **condition** node. An example of a while loop is depicted in Figure 39. The **condition** node holds a data flow graph that describes the loop expression. The two outgoing branches are marked with the labels **true** and **false**. The **condition** node determines whether the loop is to be executed by comparing the value of the conditional expression against the two constant values. If the loop is to be executed again, i.e., the conditional expression evaluates to **true**, then the **true** branch is taken. If it evaluates to **false**, then the loop execution is completed and the **false** branch to the statement-block of the next statement is taken.

4.4.10 The Infinite Loop

The third type of loop supported by VHDL is the simple loop, also called the infinite loop. It specifies a repeated execution of a sequence of statements. The syntax of an infinite loop is as follows:

loop

sequence-of-statements;

end loop;

while *condition* **loop**
sequence-of-statements
end loop;

Figure 38: VHDL While Loop

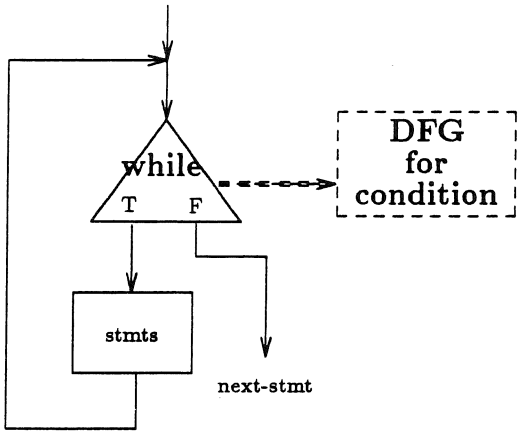


Figure 39: CDFG Representation of the While-Loop

The loop construct causes an infinite loop, unless used in conjunction with an **exit** statement. The **next** or **exit** statements are two statements that can appear in any of the three types of loops. They have the following syntax:

next [loop-label] [**when** condition];

exit [loop-label] [**when** condition];

A next statement completes one iteration of the enclosing loop by advancing control to the next iteration. The exit statement completes the execution of the whole enclosing loop.

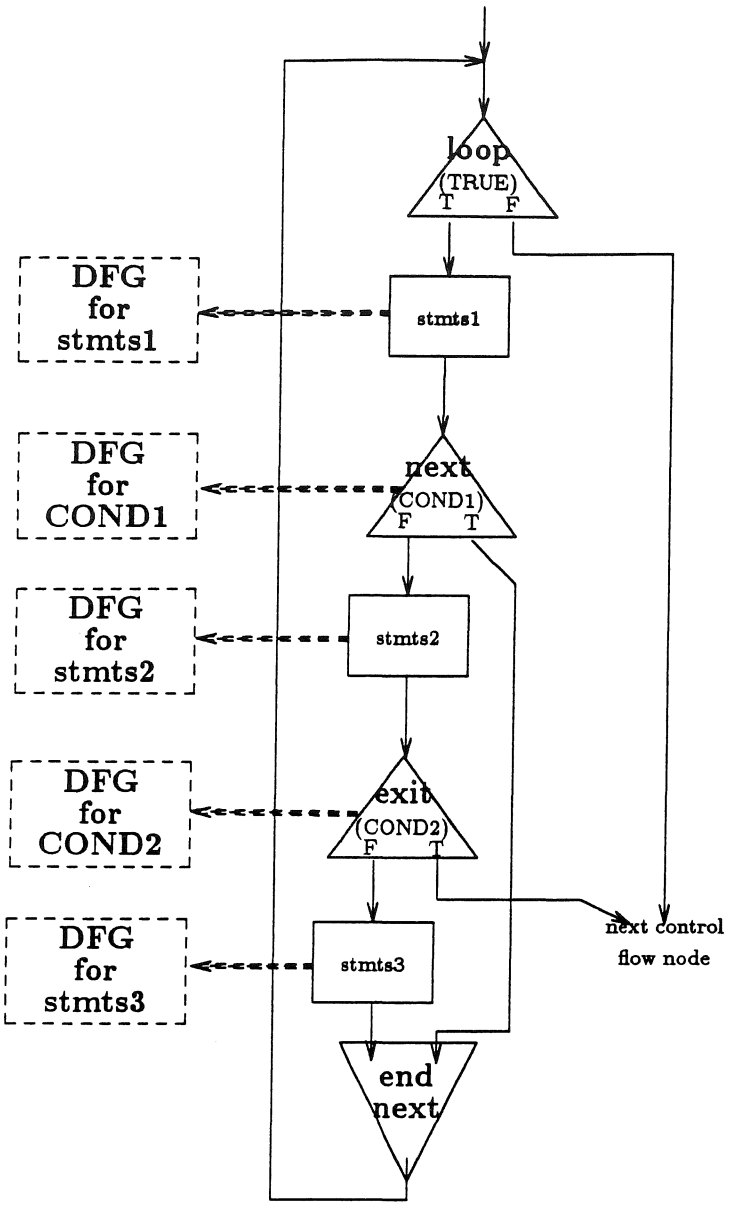
If the optional "**when** condition" clause is not present, then both constructs are modeled by control flow arcs. The **next** statement corresponds to a control flow arc that points to the end of the loop body. The **exit** statement is represented by a control flow arc that leads out of the loop to the **false** branch of the loop condition. If a "**when** condition" clause is specified, then both constructs are represented by a **condition** node.

Figure 40 depicts the design representation of an infinite loop with a next statement and an exit statement. Both statements have a "**when** condition" clause. The **next** statement is represented by a **condition** and an **end-condition** node pair. The condition node is labeled with the term **next**. The **true** branch of the condition is connected to the last statement of the loop body whereas the **false** branch is connected directly to the next statement in the loop body. The **exit** statement exists the loop indicated by the loop-label. It is represented by a condition node labeled with the term **exit**. Its **true** branch is connected to the first statement after the loop. The **false** branch of the exit statement is connected to the next statement in the loop body.

4.4.11 The Generalized Condition Node

During the previous sections, we have assumed that the **condition** node used to model a conditional or a loop statement is based on a single temporary condition variable. The CDFG model provides a more generalized **condition** node type in order to support design optimizations on the design representation. The condition listed in the condition node can be of a more general format:

- a list of one or more condition variables (and associated data types), and
- each condition variable that represents a complex conditional expression is replaced by a temporary conditional variable.



VHDL description:

```
loop
    { stmts1; }
    next when COND1;
    { stmts2; }
    exit when COND2;
    { stmts3; }
end loop;
```

Figure 40: CDFG Representation of the Infinite Loop

```
if (IN1=0 or IN2≤100)
   { A; }
else
   case I:
      10: { B; }
      20: { C; }
      others: { D; }
   end case;
end if;
```

Figure 41: VHDL Specification of Nested Conditions

The latter is accounted for in our model by associating a data flow graph with each condition node. The conditional expression that evaluates to a condition variable is represented in that graph. The result of the conditional expression is assigned to the temporary conditional variable listed in the condition node.

The condition values associated with the outgoing branches of the condition node have the following characteristics:

- they are composed of one or more condition values (or-ed),
- each condition value is a list of constant values (and-ed); one for each condition variable attached to the condition node,
- each constant is either a value from a discrete domain corresponding to the data type specified for the condition variable, a special **don't care** or **others** symbol.

To demonstrate the usefulness of an extended condition node concept, we show one design transformation in Figure 42. This design transformation can be used to optimize the design representation. In this example, a nested condition of depth two is combined into one complex condition. In the CDFG model, this is represented by an extended condition node. The condition values of this node are composed of two constant values each; one for the first and one for the second condition. The first condition value consists of one product composed of two terms, the constants T and X. This corresponds to the condition (COND1=T and COND2=**don't care**).

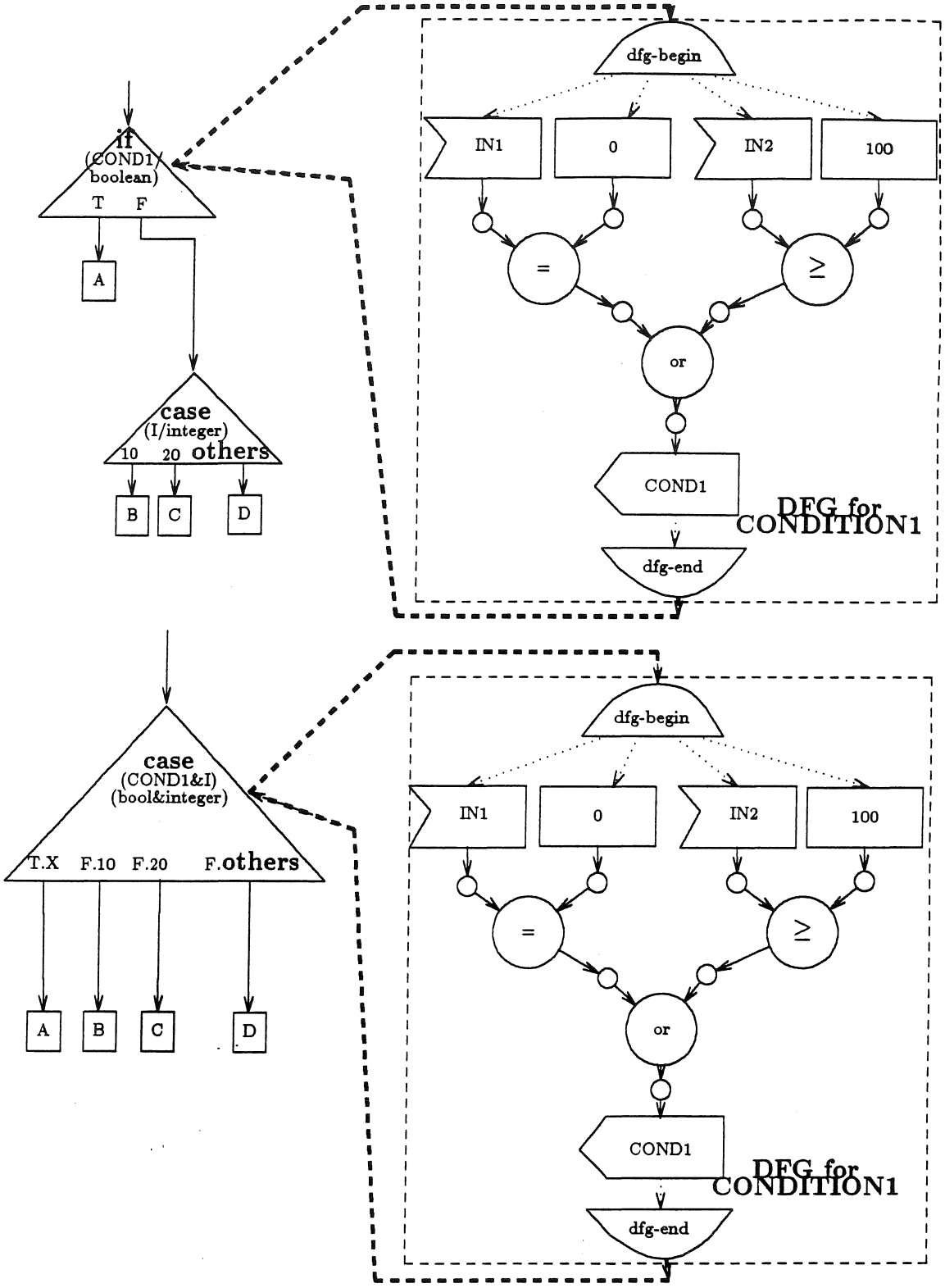


Figure 42: CDFG Representation of the Extended Condition Node

4.4.12 Timing Constraints

We support two types of timing constraints at the control flow graph level, namely, delay constraints and timeout constraints. A delay constraint enforces that the execution of a set of computations is done within a given time constraint in order to guarantee the correctness of the design. The delay constraint is used in particular by scheduling tools, which determine in which order the operations are executed and also which operations to group together into one state. A timeout constraint, on the other hand, states that the execution of the computation will resume at a different location in the control flow graph if the required timeout constraint is not met. Thus, the later does not enforce that the constrained set of computations actually completely executes within the given timing constraint.

In VHDL, a wait statement can be used to express a timing constraint within a sequential program, such as, a process or a subprogram statement. The general syntax of a VHDL wait statement is shown in Figure 43.

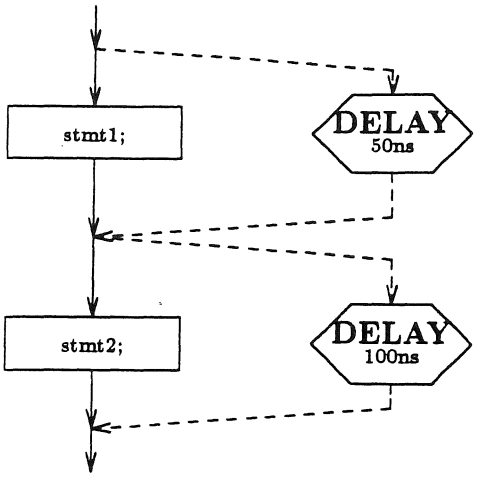
---

wait
[**on** < *sensitivity − list* >]
[**until** < *boolean − expression* >]
[**for** < *time − expression* >];

Figure 43: VHDL Wait Statement

A wait statement causes the suspension of the process or the procedure statement in which it resides. The process is resumed when an event on one of the signals in the sensitivity-list occurs and the specified condition is true. The "**for** < *time − expression* >" clause specifies the maximal amount of time the process will be suspended. This is also referred to as the maximal timeout interval. The process will resume execution at the latest after the timeout interval expires, even if the other two constraints are not met.

If a wait statement is specified with a for-clause only then we interpret this statement to be a delay from the previous wait statement (anchor point) to the current control flow node. Figure 44 represents the design representation of such a wait statement in the CDFG model. A timing constraint node with the label **delay** is inserted in the control flow graph to represent the statement. The timing arc that points to the delay node has its source in the control flow sequence arc following the previous



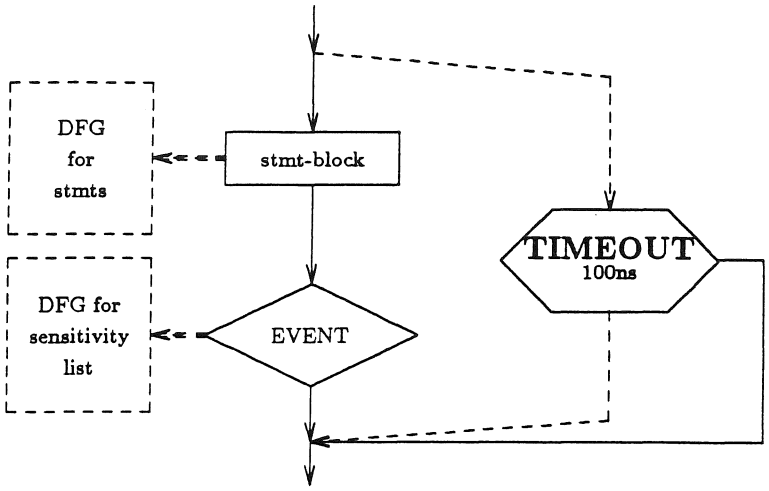
VHDL description:
wait;
stmt1;
wait for 50ns;
stmt2;
wait for 100ns;

Figure 44: CDFG Representation of the Wait-For Construct

wait statement. The timing arc originating from the delay node has its sink in the control flow sequence arc that represents the location of the wait statement.

If a wait statement combines both timing and event testing, then we interpret the timeout clause to represent an actual timeout. That is, execution continues after the specified timeout interval expires even if the event associated with the wait statement has not been fulfilled. In this case, the timing constraint node used to represent the timing constraint in the control flow graph is labeled with the label **timeout**. The timing arcs embrace the event node that represents the current wait statement. Figure 45 represents the design representation for this wait statement type.

If there is a single wait statement in the graph, then we require the wait statement to be the last statement of the process. In this case, the timing arcs will start from the first node and end with the last node of the control flow graph.



VHDL description:
wait;
stmts;
wait on < *sensitivity – list* > **for** 100ns;

Figure 45: CDFG Representation of the Wait-On-For Construct

Note that we support timing constraints at both the control flow and the data flow graph level. The timing constraints at the data flow graph level are generally of a finer granularity, i.e., they restrict the execution time of one or several operation nodes. Whereas the timing constraints at the control flow graph level are of a coarser granularity, i.e., they refer to a complete data flow graph at a time. Thus, the timing constraints at the control flow graph level are mostly used by the scheduler to determine the ordering of states, while the timing constraints at the data flow graph level are used by hardware allocators to determine which unit to assign to a flow graph node. We make the assumption that the timing constraints specified at the data flow graph level are consistent with those at the control flow graph level. For example, if an operation node within a data flow graph has a minimal delay of 20ns, then the complete data flow graph cannot have a maximal delay of only 10ns. Such an inconsistent specification would simply be rejected by the design tools as an incorrect design specification.

4.4.13 Asynchronous Events

An event node is used to model asynchronous events at the control flow graph level. In an asynchronous design, the execution steps are initiated based on the occurrence of particular events rather than based on the regular clock pulse. An event node has an associated data flow graph that describes the event condition. When the event specified by an event node occurs, then execution continues with the control flow node after the event node.

Asynchronous events can be expressed in VHDL by a wait statement that contains an event-clause, a condition-clause or both. The syntax of these statements is:

wait on $< sensitivity - list >$;

wait until $< boolean - expression >$;

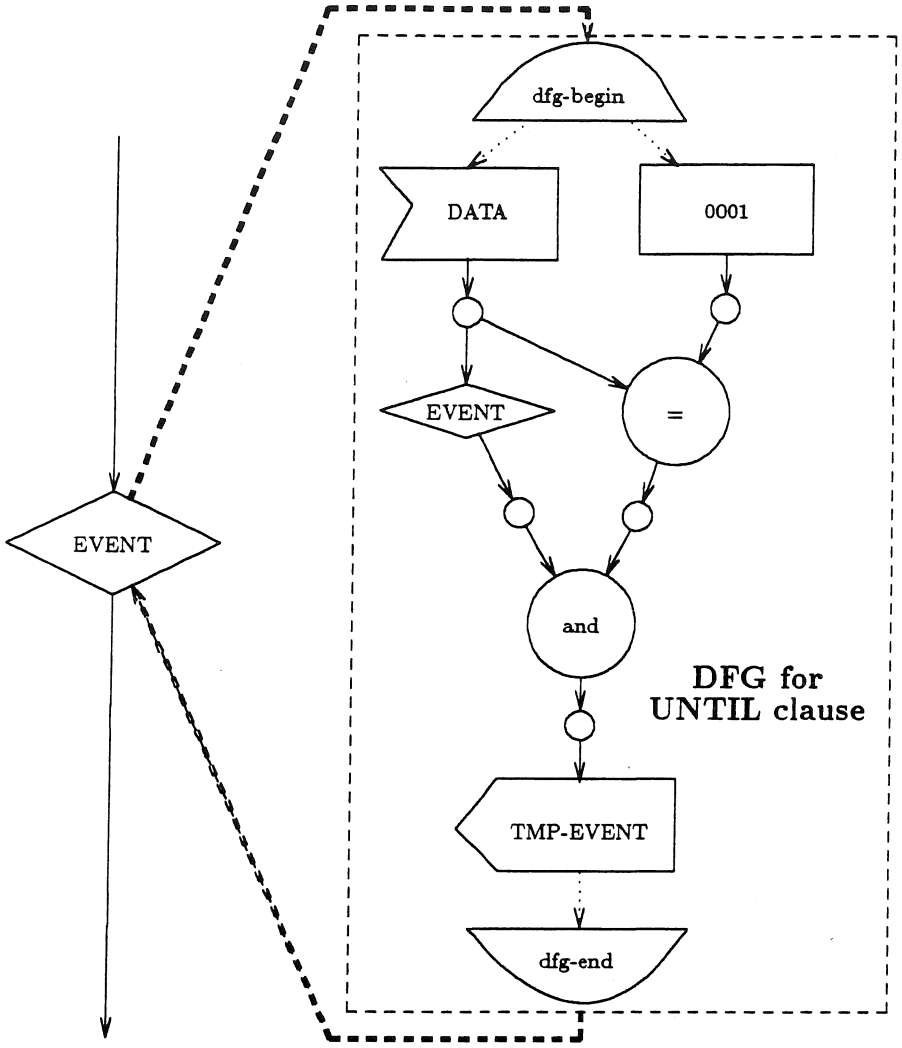
wait on $< sensitivity - list >$ **until** $< boolean - expression >$;

The **on**-clause specifies a list of signals to which the process is sensitive. The **until**-clause specifies a condition that must be true before the process can be resumed. We model both cases by an event node in the control flow graph. The event node has an associated data flow graph that describes the event.

Figure 46 depicts the design representation for a wait statement with an **until**-clause. Since the wait statement does not contain an explicit sensitivity list, all signals used in the **until**-clause are inserted into an implicit sensitivity list. In Figure 46 this is represented by inserting an event node in the data flow graph that tests whether an event has occurred on the signal DATA.

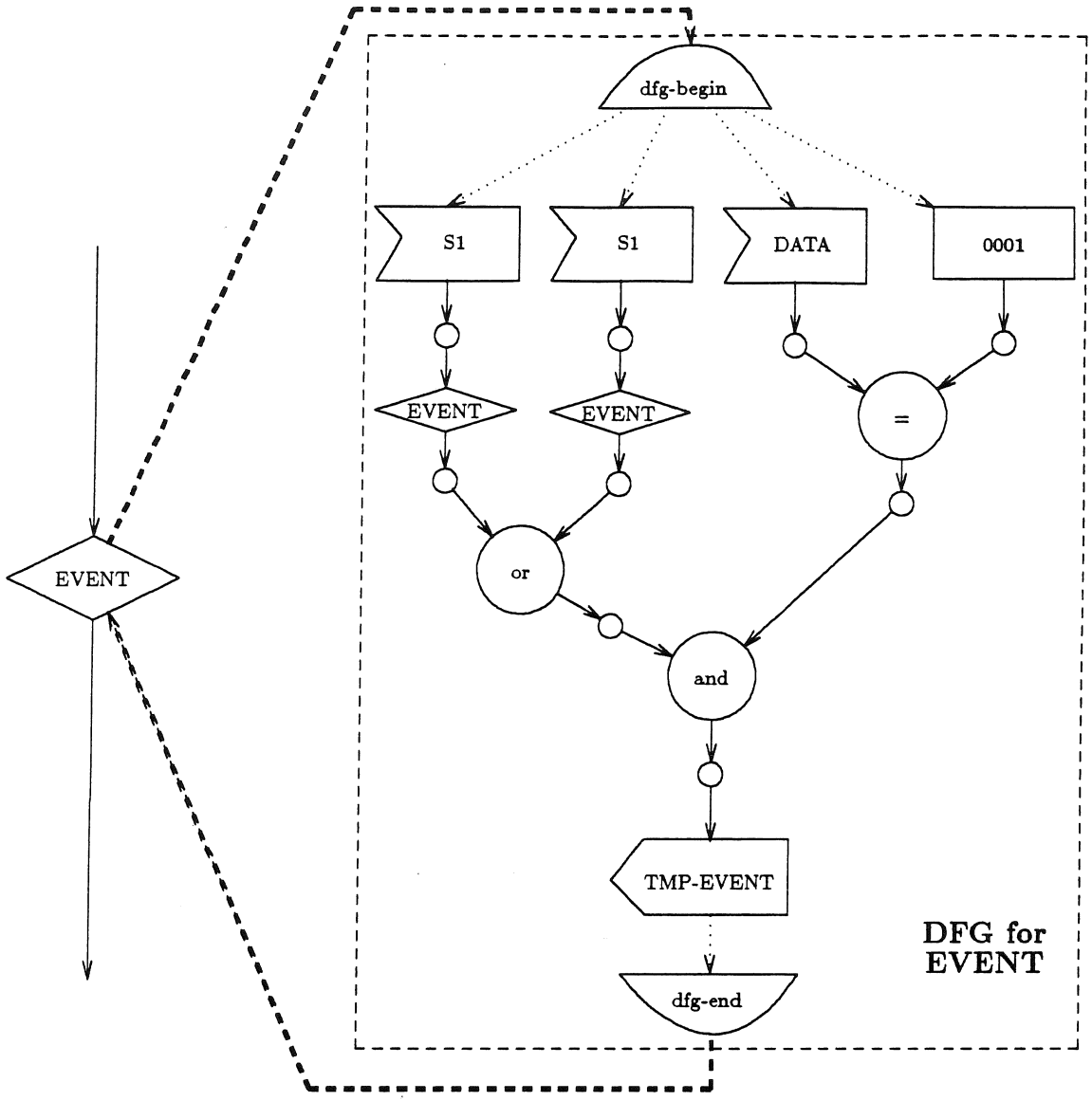
Figure 47 depicts the design representation for a wait statement that contains both an **on**-clause and an **until**-clause. In this case, the wait statement has an explicit sensitivity list; therefore, the signals mentioned in the **until**-clause are not added to the sensitivity list. The data flow graph that describes the **event** does hence not test whether an event has occurred on the signal DATA.

For an example of the representation of a wait statement that contains an **on**-clause but not an **until**-clause see the left hand side of Figure 47. It corresponds to the representation of the VHDL wait statement "**wait on** S1, S2;". In general, the CDFG representation of this wait statement contains an event node in the data flow graph for all signals in the **on**-clause. The outputs of these event tests are anded together. The result of this test is stored in a temporary variable, called TMP-EVENT.



VHDL description: **wait until** (DATA="0001");

Figure 46: CDFG Representation of the Wait-Until Construct



VHDL description: **wait on** S1, S2 **until** (DATA="0001");

Figure 47: CDFG Representation of the Wait-On-Until Construct

4.4.14 Process Sensitivity List

An alternative method for describing an event in VHDL is a process *sensitivity list.* This is a list of one or more signals of the enclosing environment of a process that is inserted into the header statement of a VHDL process. Semantically a process with such a list contains an implicit wait statement as the last of its statements. The execution of the process will be suspended when the end of its sequence of statements is reached. The process is resumed when an event occurs that changes one of the signals in the sensitivity list.

We model this by inserting an event node at the end of the control flow graph that represents the process. The event node has an associated data flow graph that tests for a change on the signals listed in the sensitivity list. This event node is similar to the one shown in Figure 47. Note that a process with a *sensitivity* list cannot have any explicit wait statments within its body. Therefore, there will be only one event node in the corresponding control flow graph representation of such a process.

5 THE STATE TRANSITION GRAPH

5.1 Integrating State Information into the CDFG Model

High-level synthesis tools, in particular, state schedulers, slice the control/data flow graph into states. This state information leads to the generation of a state table that represents the sequencing of the design over time. This state table is then compiled by a control compiler into control logic. State scheduling information will be captured in DDM in two ways. First, the slicing of the behavioral description into states is represented by annotations to the CDFG graph. Secondly, the encoding of the state table into control logic results in the creation of a component, called the control unit. This unit is inserted into the register-transfer level component graph. In this section, we are concerned with the former, the state information at the control/data flow graph level.

There are several routes for the representation of the state information in the CDFG. We first present possible alternatives and then discuss their pros and cons. This constitutes a motivation for the approach we have decided to take.

Possible Approaches:

1. Associate state information with each data flow node.
2. Associate state information with each control flow node.
3. Associate state information with each control flow node and with each data flow node.
4. Create a separate list of state nodes and state sequencing arcs (a state transition graph) and associate control flow nodes with these state nodes.
5. Create a separate list of state nodes and state sequencing arcs. Associate control flow nodes with their corresponding state node and also annotate data flow nodes by state information.

Approach One. Solution one is what comes to mind first, since scheduling is concerned with assigning data flow operations to particular states. There are however several problems with this solution. First, a data flow graph imposes only a partial order on its nodes. Therefore, it will not necessarily be obvious in which order the states are to be executed. In addition, the overall structure of the state sequencing is less apparent. Information pertaining to a state, such as the state

transition conditions or the representation of events that trigger the transition into the next state is not easily incorporated into this scheme.

Approach Two. The next logical solution is to associate the state information with the control flow rather than with the data flow nodes. The control flow graph represents the sequencing of the behavior over time in the form of a partial order. State assignments further refine this partial order to a complete order with the additional restriction of fixed time intervals. Some control flow nodes, in particular, statement-block nodes, which capture potentially big chunks of sequential code may be mapped into several states. Therefore, a control flow node would have to be annotated by more than one state identifier. Thus, the scheme of simple control flow node annotations would not be fine-grained enough to establish a total order. Another disadvantage of this approach is that information pertaining to an individual state is again spread over several control flow nodes.

To remedy the ambiguity problem, a statement-block node and its associated data flow graph could be broken into a sequence of statement-block nodes. Then each portion of the data flow graph assigned to one state would be grouped in a separated data flow graph. This guarantees that all data flow information associated with a statement block node is assigned to the same state. One problem that arises from this separation of a data flow graph into multiple small data flow graphs is the representation of multi-cycle operation nodes. A multi-cycle operation node is active over several cycles (states). It reads in the same values in all states and writes out results only in the last state. It is awkward to represent this situation if the activation duration of an operation node is broken into several data flow graphs. One would need to distinguish between an operation node that is being executed several times within consecutive states and a multi-cycle node that is been executed once but that is active over several ones. In addition, loss of information may have occurred during the control flow graph reorganization. For instance, since data flow arcs do not cross data flow graph boundaries, control dependency arcs as well as timing arcs at the data flow graph level could have been removed. This would make changes in the schedule, such as, increasing the clock cycle time by a few nano seconds, difficult, since they may require the regrouping of data flow nodes among the different statement-block nodes.

Approach Three. The third solution solves the problem of the second approach in as much as a data flow graph does not have to be divided into multiple data flow graphs to reflect the state structure. The representation of multi-cycle operation nodes could be addressed by assigning a sequence of state identifiers rather than a single state identifier to such a node. The last disadvantage of solution two is, on the

other hand, not addressed by this approach. Information about one state is spread over several control flow nodes.

Approach Four. The fourth approach addresses the critique of spreading information specific to a state over several places, and possibly redundantly duplicating it. This solution stores all information specific to a state into the corresponding state node. This solution still suffers from the problems of the second approach: One control flow node may be assigned to more than one state. Therefore, a control flow node would have to be split into several control flow nodes to maintain which data flow information belongs to which state.

Approach Five. The fifth approach, being a superset of the fourth approach, has the same advantages as the fourth approach. That is, it addresses the critique of isolating information specific to a state, such as, the state name, the number of successor states, and the predecessor states. In addition, statement-block control flow nodes do not have to be separated into several control flow nodes. Thus, the fifth approach deals successfully with problems of the second solution. Multi-cycle operation nodes can conveniently be represented by assigning a sequence of state identifiers to them. The fifth approach requires some more storage space; the space overhead is in the order of the number of states.

For the reasons given in this section we opt for the fifth approach. We thus propose to extend the CDFG structure by a state transition graph (STG) structure. Such a state graph shows the sequencing of the design over time by associating control flow nodes with their corresponding state node. In addition, we extend the CDFG model by one additional set of attributes, namely, data flow nodes are annotated by a state information attribute. More precise definitions of the state transition graph are given next.

5.2 State Transition Graph Definition

Below, we first define the state transition graph and then discuss its constructs in more detail.

Definition 4 *A* **state transition graph** *is a directed (possibly cyclic) graph STG = (STN,STE,STF) with STN the set of vertices and STE the set of edges. The elements of STN are uniquely identified by the function* **state-name**: $STN \rightarrow INTEGER$.

1. *STN, the set of state nodes, is composed of three disjoint sets, STN = state-nodes* $\cup$ *concurrent-state-nodes* $\cup$ *hierarchical-state-nodes.*

- *State-nodes represent the states of a state automata.*
- *Concurrent state nodes are used to demark concurrent subprocesses.*
- *Hierarchical state nodes describe the decomposition of a state into sub-states.*

2. *STE, the set of state edges, is composed of three disjoint sets, STE = state-transition arcs ∪ state-hierarchy-arcs ∪ state-to-control-flow-arcs.*
   - *State-transition arcs sequence between two states.*
   - *State-hierarchy arcs link a state node to the state transition graph it is decomposed by.*
   - *State-to-control-flow arcs link a state transition graph to the corresponding control flow graph.*
3. *STF is a set of state transition marking functions STF_i: STN → STM with STM the set of possible attribute domains.*

5.3 Representation of the State Transition Graph Nodes

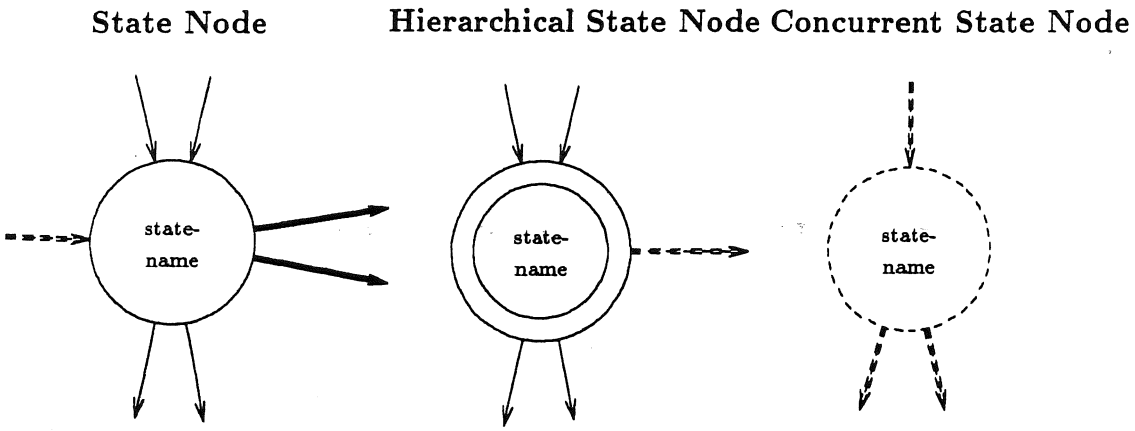


Figure 48: Graphical Representation of Nodes in the State Transition Graph

The graphical depictions of state transition graph constructs are shown in Figure 48 and 49. Each state transition graph vertex is explained in more detail below, while the state transition graph arcs are described in the next section.

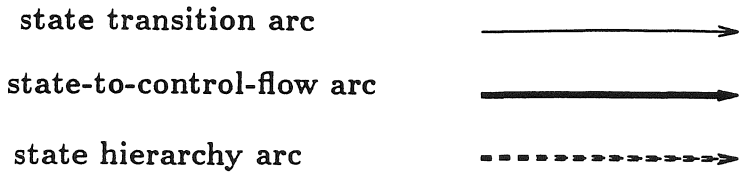


Figure 49: Graphical Representation of Arcs in the State Transition Graph

Type: State node

Graphic: Circle with state name

Description: A state node models a state, which could be either a super-state or a simple state. A state node has one or more incoming and one or more outgoing transition arcs to other state nodes within the state transition graph. The outgoing transition arcs are mutual exclusive. At any point of time, exactly one of them will be selected based on conditions that are captured in the associated control flow graph. A state node has or or more state-to-control-flow arcs which point to the control flow nodes that are assigned to the state. A state node may optionally have an incoming hierarchical state arc; if it is part of a (sub-) state transition graph that describes a higher-level hierarchical or concurrent state.

Type: Hierarchical state node

Graphic: Dashed circle with state name

Description: A hierarchical state node models hierarchy in the state transition graph. A hierarchical state node describes how one state is decomposed into a number of sub-states. It has exactly one outgoing hierarchy arc that points to the first state node in the underlying (sub-) state transition graph. A hierarchical state node has one or more incoming and one or more outgoing transition arcs to other state nodes in the current (super-) state transition graph.

Type: Concurrent state node

Graphic: Double circle with state name

Description: A concurrent state node is a demarcation node for concurrent state transition graphs. Therefore, a concurrent state node must have two or more outgoing hierarchy arcs to other state transition graphs, i.e., to the first state nodes of concurrent sub-state transition graphs. A concurrent state node has one incoming state hierarchy arc from a hierarchical state node in the super-state transition graph. The concept of concurrency thus includes the concept of hierarchy at the state transition graph level.

5.4 Representation of the State Transition Graph Arcs

The graphical symbols of state transition graph arcs are depicted in Figure 49. Each arc type is explained in more detail below.

Type: Transition arc

Graphic: Arrow

Description: A state transition arc connects two state nodes within the same state transition graph. It thus shows the sequencing of states.

Type: State-to-control-flow arc

Graphic: Bold Arrow

Description: A state-to-control-flow arc connects a state node to control flow nodes. It is a demarcation device that lists all control flow nodes that have been assigned to the same state. There may be more than one control flow node associated with one state. Also, one control flow node may be associated with more than one state. Hence, this models an many-to-many relationship.

Type: State-hierarchy arc

Graphic: Dashed Arrow

Description: A state-hierarchy arc connects a hierarchical state node to the state transition graph into which it is decomposed. Thus, a state-hierarchy arc points from a hierarchical state node either to the first state node of another state transition graph or to a concurrent state node. Similarly, a state-hierarchy arc is also used to connect a concurrent state node to two or more concurrent sub-state transition graphs.

5.5 Discussion

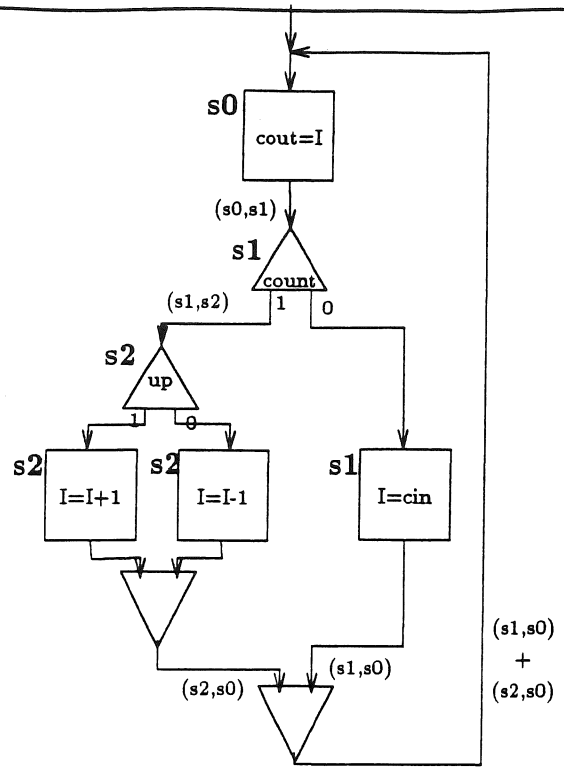
A state node can model either a **simple-state** or a **super-state**. If a state is of type super-state, then the state will be further refined into two or more states before a control unit can be created. If it is of type simple-state, then it denotes a state of the current state transition graph that can be executed within the given clock cycle. The state will thus not further be decomposed into smaller states. A super-state node tends to point to many control flow nodes while a simple state node generally points to only one or two nodes.

If there is more than one state transition leaving a state node, then the state transitions are conditional. There are one or more conditions in the control flow graph that specify which state transition arc is chosen. These conditions can thus be derived from the control flow graph; they are however not explicitly represented in the state transition graph. If the design is asynchronous, these transitions may depend on an event instead of the clock pulse. Similarly, the events that trigger the transition to the next state are captured in the control flow rather than the state transition graph.

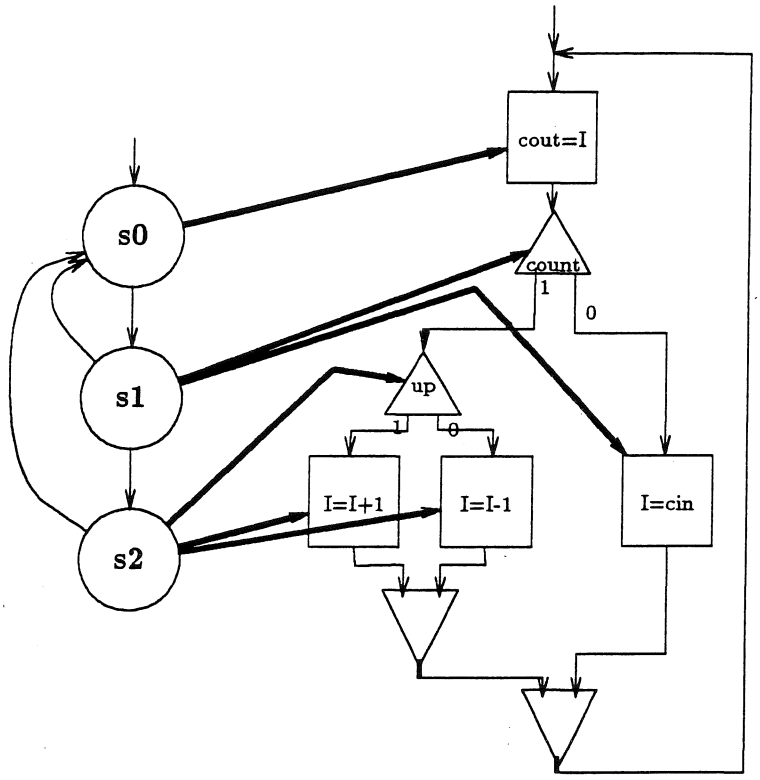
Next we present an example of a control/data flow graph with state assignments. In Figure 50.a, we show how the state assignments may conceptually be viewed by a user of DDM. Figure 50.b, on the other hand, shows how this state information is kept within the CDFG model. In this example we see that some states, i.e., state S1 and S2, contain more than one control flow node. Also some control flow nodes may not be assigned to any state at all. The latter is true for control flow nodes which don't contain any data flow information, such as, **end-condition** nodes or demarcation nodes. The **end-condition** node marks the end of a set of mutual exclusive conditional execution paths. It is a collection point for several control flow sequence arcs.

5.6 State Information Extension to the CDFG

The links from the state transition graph to the control flow graph discussed in the previous section are assumed to be bi-directional. Therefore, we extend the control flow graph as presented in Section 4 in the following manner. Each control flow node has one additional reference called **assigned-state.** It represents a reference to all states that a control flow node has been assigned to. In most cases, this will be one individual state. For control flow nodes of type statement-block, however, there may be more than one associated state. This is so since a data flow graph represented by one statement-block control flow node may be sliced into more than one state.



a. control flow graph annotated with state information



b. representation of annotated control flow graph

Figure 50: CDFG of Counter Example with State Assignments

In addition, we extend the underlying data flow graph model presented in Section 3 in the following manner. Each data flow node has one additional attribute type called **assigned-state**. This attribute denotes the identifier of the state that the data flow node has been assigned to. If the data flow node is a multi-cycle operation node then this attribute is a list of states identifications rather than a single state. This list of states is required to consist of states that are consecutive in the state transition graph.

6 The ANNOTATED COMPONENT GRAPH

A register-transfer level structure is represented by a set of components and their connections. This is commonly referred to as a netlist. A formal definition of the representation of a such netlist, called the **annotated component graph model**, is given next. More details on the modeling constructs are presented thereafter.

6.1 Definition of the Annotated Component Graph

Definition 5 *An* **annotated component graph** *is a directed graph* $ACG = (CGN, CGE, CGP, CGF)$*:*

1. *CGN corresponds to the set of vertices of the annotated component graph. We distinguish between component nodes, port nodes, net nodes, and decomposition nodes.*

2. *CGP, the set of pins, corresponds to the connection points of vertices with the arcs of the graph. The function* **pin-class**: $P \rightarrow \{$*input-pin, output-pin, in-out-pin*$\}$ *specifies whether a pin is an input pin, an output pin or both. A vertex* $v \in CGN$ *can have an ordered possibly empty list of input and output pins, respectively. The function* **input**: $CGN \times INTEGER \rightarrow CGP \cup \emptyset$ *is an assignment of input pins to vertices. The function* **output:** $CGN \times INTEGER \rightarrow CGP \cup \emptyset$ *is an assignment of output pins to vertices.*

3. *CGE represents the set of directed edges between the (pins of the) vertices of the annotated component graph. The edges correspond to pairs* $(p1, p2) \in CGP \times CGP$ *with the direction of the edge from p1 to p2. We distinguish between three types of edges, which are interconnection arcs, hierarchy arcs, and demarcation arcs.*

4. *CGF is a set of data flow marking functions* CGF_i*:* $CGN \rightarrow CGM$ *with CGM the set of possible attribute domains.*

6.2 Representation of the Nodes in the Annotated Component Graph

The graphical depictions used to represent the constructs of an annotated component graph are shown in Figures 51 and 52. Similarly as in the CDFG graph, the connection points, here called pins, of a vertex with an edge are not explicitly represented.

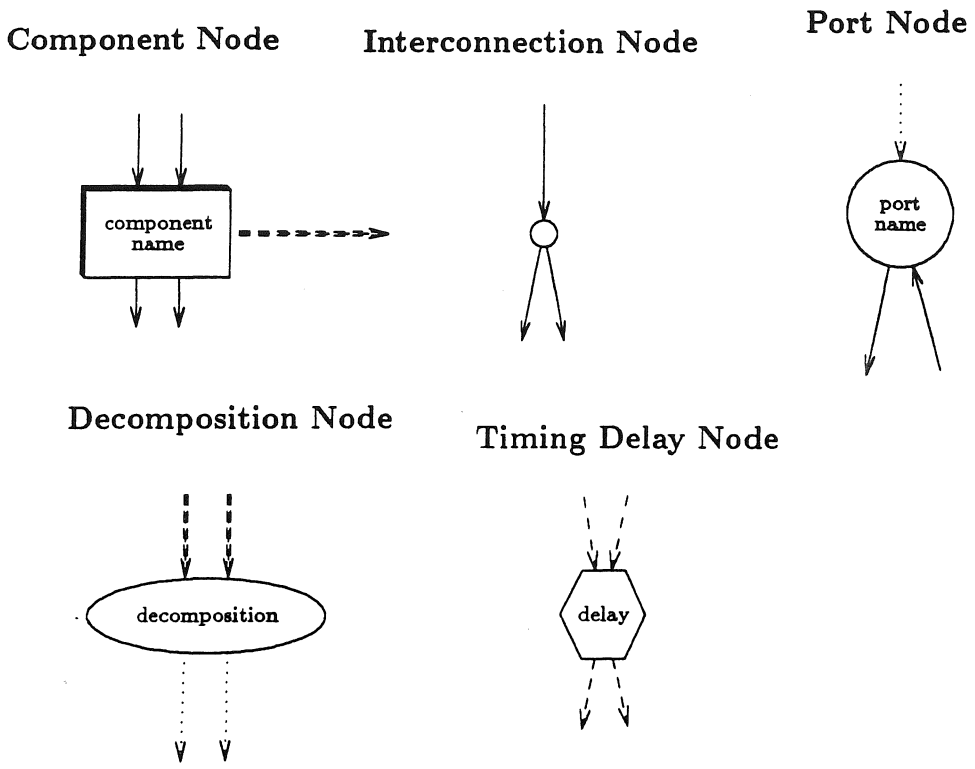


Figure 51: Graphical Representation of Nodes in the Annotated Component Graph

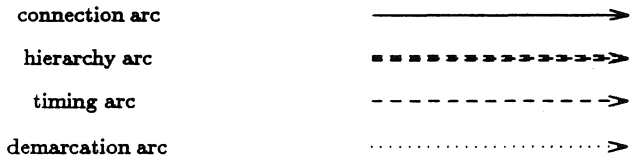


Figure 52: Graphical Representation of Arcs in the Annotated Component Graph

Instead, the connection arcs directly connect two component nodes. The meaning and attributes of each vertex type is explained next.

Type: Component node

Graphic: Box with heavy line with the component name

Description: A component node represents a component. It can for instance be a functional unit, a multiplexor, or a bus. It can also be any other non-primitive unit that is further decomposed by a hierarchical description. A component node conceptually corresponds to a set of pins, which mark its connection points with other components.

Type: Port node

Graphic: Circle with the port name

Description: A **port** node is a special type of component that represents the input and output communication points of an annotated component graph with its environment. If an annotated component graph is encapsulated into one super-component, then the port nodes correspond to the pins of the super-component.

Type: Interconnection node

Graphic: Small Circle

Description: An interconnection node is used to model an interconnection net between two or more components. Each interconnection node has one input pin and one or more output pins. These pins are connected to connection arcs. They represent the source and the destinations of net. An interconnection node represents the media on which a data value travels through the structure, i.e., it corresponds to a wire. Consequently, it can only hold one data value at any point in time and therefore has only one source.

Type: Decomposition node

Graphic: Ellipsis

Description: A **decomposition** node is used to demark an annotated component graph that represents the decomposition of one super-component node into a structure of more primitive components. The pins of the higher-level component are represented explicitly in the decomposition ACG graph as port nodes. They mark the connection points of the higher-level component and the more primitive components out of which the former is composed of. There is exactly one port node for each pin

of the super-component. These port nodes are attached to the **decomposition** node by demarcation arcs.

Type: Timing node

Graphic: Stop sign symbol

Description: A timing node models a timing constraint. A timing node has one or more incoming timing arcs and one outgoing timing arc. The incoming timing arcs connect the delay node with the sources of the delay while the outgoing timing arc points to the destination data value node. The timing node specifies a delay for the execution of all component nodes between the source nodes and the sink node of the delay. Optionally, the timing node may associate an attribute called event-type (which takes the values RISING, FALLING, and CHANGING) with its source and its sink nodes. In addition, it may give a delay value for the a minimal, nominal and maximal delay constraint, respectively.

A timing node models two types of timing constraints, which are **path delays** and **event-related** delays. A **path-delay** timing node models the time taken for the effect of a signal to propagate through a set of hardware units from one point of the hardware to another. A **path-delay** timing node is given the label **path-delay**, or short, **delay**.

By default, the path delay node constitutes a timing constraint for all component and interconnection nodes on the paths starting from the source nodes of the delay node and ending with the destination node. A delay node has an optional attribute, called **path expression**, which describes some of these paths between the sources and sink node. If a **path expression** is given, then the path delay constrains only those component nodes listed in the path expression.

An **event-related delay node** captures timing relationships between the occurrences of possibly independent events, like, for instances set-up and hold times. An event-related delay node is given the label **event-delay**, or short, **event**.

6.3 Representation of the Arcs in the Annotated Component Graph

The graphical depictions for the arcs in the annotated component graph model are given in Figure 52.

Type: Connection arc

Graphic: Arrow

Description: A connection arc corresponds to a wire. It represents an interconnection between a component node and a net node by connecting an output pin of a component node to an input pin of an interconnection node, or, vice versa.

Type: Hierarchy arc

Graphic: Bold Dashed Arrow

Description: A hierarchy arc associates a non-primitive component node with the primitive components out of which it is composed. Thus, it connects the component node with its associated decomposition node.

Type: Demarcation arc

Graphic: Dotted Arrow

Description: A demarcation arc connects the ports of a component node to its **decomposition** node.

Type: Timing arc

Graphic: Dashed arrow

Description: A timing arc connects a timing constraint node with an interconnection vertex. The set of timing arcs associated with a timing constraint node mark the group of component and interconnection nodes that are constrained by the timing constraint node.

6.4 A More Detailed Description of the Annotated Component Graph

The annotated component graph (ACG) describes the structure and geometric layout of a design. Rather than distinguishing between many different structural vertex types, we define various attribute domains for them. Below, we discuss these attributes.

6.4.1 Timing Constraints

Timing constraints in the annotated component graph are very similar to timing constraints in the data flow graph. Therefore, for a detailed discussion of timing

constraint nodes and their attributes the reader is referred to Sections 3.4.15 and 3.4.16.

Some of the timing constraints in the data flow graph may be carried over to the annotated component graph. For instance, if a setup delay is specified for a write-node in the data flow graph, then this setup delay will also be specified for the register to which this write node has been mapped to in the annotated component graph. Other delays are either completely omitted or they are broken up into several smaller delays by the synthesis tools. For instance, if a delay is specified for a set of data flow operations that are scheduled into two or more states, then this delay can no longer be preserved in the annotated component graph. This delay will, on the other hand, be used by the scheduling tool to determine which of the operation nodes can be put in the same state. Essentially, the initial delay is compiled into the allowed duration of a state (the clock cycle), and thus is no longer needed by synthesis tools once scheduling is completed.

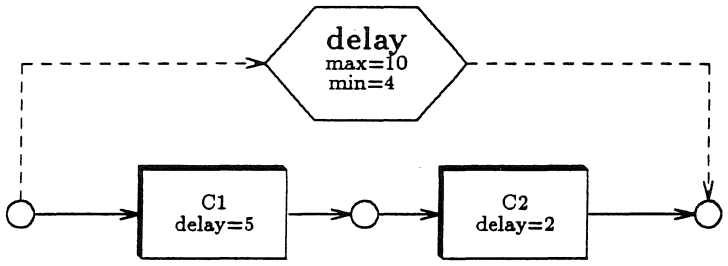


Figure 53: Timing Constraints in the Annotated Component Graph

Each component node also carries timing attributes as discussed in a later section. These attributes are not constraints on the component, rather they are characteristics of the chosen component. In Figure 53, a timing constraint for the paths through C1 and C2 is specified with a minimal delay of 4ns and a maximal delay of 10ns. The timing attributes associated with the components C1 and C2 correspond to the actual delays of the instantiated components. C1 and C2 have a delay of 5ns and 2ns, respectively. Thus, a path through both will have a delay of approximately 7ns. It can be verified that the timing constraint of 10ns is met. The maintenance of the timing constraint information is particularly useful for optimizations on the structure graph. A logic optimization tool can for instance replace one or both of these two components by another component type with other timing attributes as long as the timing constraint is met.

6.4.2 Structural Attributes of Component and Interconnection Nodes

A structure is represented by a set of components and their connections. The **component type** of a component can either be complex or primitive. A component is called primitive if it is not further decomposed into subcomponents. A primitive component is either an instantiation of an instance of a generic component library or it corresponds to some actual hardware unit. Each primitive component has a set of attributes describing its component type. Example attributes are the component class attribute and the function attribute.

A component is called complex if it is further decomposed into subcomponents. Some of the attributes of primitive components, such as, a list of functions they execute, may be undefined for them. A complex component is further defined by an annotated component graph. It has an associated decomposition node which lists the ports of the component node as well as the annotated component graph composed of the more primitive subcomponents. A decomposition node is a demarcation node that describes the hierarchical composition of a component into lower-level components. A decomposition node has therefore no equivalent hardware construct.

A **component** node has the information associated with it:

- component name,
- component type: primitive, complex,
- number of input pins,
- number of output pins, and
- a list of pin information (see below).

The interconnection points of a component node, called pins, are described by the following:

- unique pin name,
- pin class: input, output, and input-output,
- pin type: control, data, clock, set, reset, enable, and select, and
- bit width of the pin.

If a component node is primitive, then it has the following additional attributes:

- component class: register, memory, functional-unit, bus, multiplexor,
- file name of its IIF description generated by ICDB, and
- functions: such as, add, sub, etc.

The control unit is inserted as component into the annotated component graph. The control lines that connect the control unit with the data path components are also inserted as connection nodes into the annotated component graph. The behavior of the control unit is stored in form of a separate behavioral description since it is random logic.

In hardware, an **interconnection node** corresponds to connections between components, i.e., a wire. The connection arcs on the other hand are needed to show the sources and sinks of a particular wire. Structural attributes are associated with an interconnection node but not with connection arcs:

- unique net name,
- bit width of the net,
- name of the input pin,
- name of all output pins, and
- net type: data, control.

Interconnection of the annotated component graph is represented by connecting the pins of the component nodes with the pins of the interconnection nodes.

6.4.3 Geometric Attributes of Component and Interconnection Nodes

The geometric domain describes the circuit's geometric layout. Generally speaking a layout consists of a set of cells and their connections. Instead of developing a separate hierarchy of cells and wires, we associate the geometric information with the corresponding structural components and nets in form of additional attributes. The following geometric attributes are associated with the component and port nodes of the annotated component graph:

- aspect ratio: a width and height pair,

- position coordinates: a x-coordinate and y-coordinate pair,
- a list of position coordinates for the input and output pins: (pin-name,x-coordinate, y-coordinate),

The aspect ratio of a component specifies its width and height. For instance, if a component has the aspect ratio (4mm,5mm) and position coordinates (1mm,1mm), then the rectangle fills a space between the four coordinates (1mm,1mm), (1mm,6mm), (5mm,1mm), and (5mm,6mm). The input and output pins of the component are positioned at the periphery of the component. Consequently, the position coordinates given for the pins are relative to the width and height of the component. For instance, a pin positioned in the middle of the top horizontal edge of the component would be described by the coordinate (3mm,6mm).

A set of structural components may be combined and treated as one component for layout purposes. For instance, a partitioning design tool may combine all random logic into one module. This requires a regrouping of the structure graph to reflect the new decomposition of the structure into components since geometric attributes are associated with individual structure nodes only.

An **interconnection node** has the following geometric attributes:

- starting position coordinate, and
- a sequence of coordinate positions.

A net may connect one source component to more than one destination component. Therefore, the geometric information will be a set of starting position coordinates and their corresponding sequences of coordinate positions. The later describe the routing.

6.4.4 Timing Attributes of the Component and Interconnection Nodes

Each component node has optional attributes that describe its input to output timing behavior. We again distinguish between two types of timing attributes: (propagation) delay and others. Propagation delay corresponds to the path-delay timing constraint. Its specification consists of the following parts:

- **source**: pin name
- **sink**: pin name

- **function**: function name
- **delay duration**: minimal, nominal, and maximal delay
- **delay value**: actual delay in nano seconds

Source and sink are references of input and output pins of the component, respectively. The delay specification can specify three different types of delays, which are, minimum delay, nominal delay, and maximal delay. They denote the delay for a signal to propagate from the indicated input pin through the component to given output pin. A component may be multi-functional, for instance, a functional unit may implement the Addition and the Subtraction operation. Therefore, the above specified delay can be specified relative to the execution of each function. If a function name is not given in a delay specification, then the delay is assumed to hold for all functions that the component implements.

The second type of timing characteristics is presented as a pair. The first item of the pair gives the name of the delay and the second corresponds to the associated delay value. For instance, for a register one may want to specify the following two delays:

(**setup time**, 10ns)

(**hold time**, 8ns)

6.5 An Example of an Annotated Component Graph

The following example is given to explain how the annotated component graph represents a hierarchical netlist structure. A structural VHDL description of a full-adder component is given in Figure 54. This example full-adder component is constructed out of three subcomponents: two half-adder components and one or-gate. The signals declared in the architecture body internally connect the subcomponents to form the structure. Thus they correspond to interconnection wires. The connection paths of inputs to outputs are specified by three *component instantiation statements.* Each such statement uses a component defined by a local *component declaration.* It creates an instance of such a component by giving an association list (**port map**) that associates actuals (like the signal TempSum) and ports (like the port CarryIn) with locals (the ports of the local component declaration).

Figure 55 shows a schematic of the full adder. Such a schematic is commonly used to depict a structural decomposition. Figure 56, on the other hand, presents the annotated component graph for the full adder design representation which demonstrates

```
entity Full_adder is
  port
    (A : in Bit;
     B : in Bit;
     CarryIn : in Bit;
     AB : out Bit;
     CarryOut : out Bit;
    )
end Full_adder;

architecture Structure_Full_adder of Full_adder is
signal TempSum, TempCarry1, TempCarry2 : Bit ;

component Half_adder
   port (X: in Bit; Y: in Bit; Sum: out Bit; Carry: out Bit );
end component;

component Or_gate
   port (In1: in Bit; In2: in Bit; Out1: out Bit);
end component;

begin
  I0: Half_adder
    port map (X=>A, Y=>B, Sum=>TempSum, Carry=>TempCarry1);
  I1: Half_adder
   port map (X=>TempSum, Y=>CarryIn, Sum=>AB, Carry=>TempCarry2);
  I2: Or_gate
    port map (In1=>TempCarry1, In2=>TempCarry2, Out1=>CarryOut);
end Structure_Full_adder;
```

Figure 54: Entity Declaration and Architecture Body of a Full-Adder

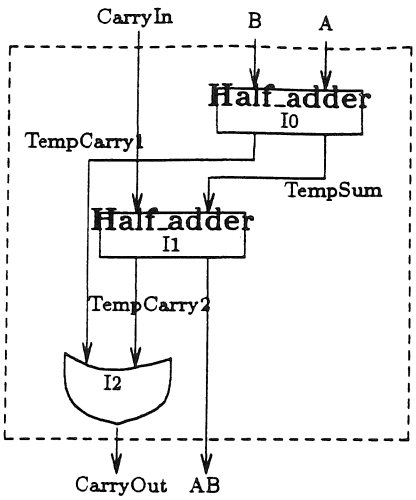


Figure 55: Block Diagram of the Full-Adder Example

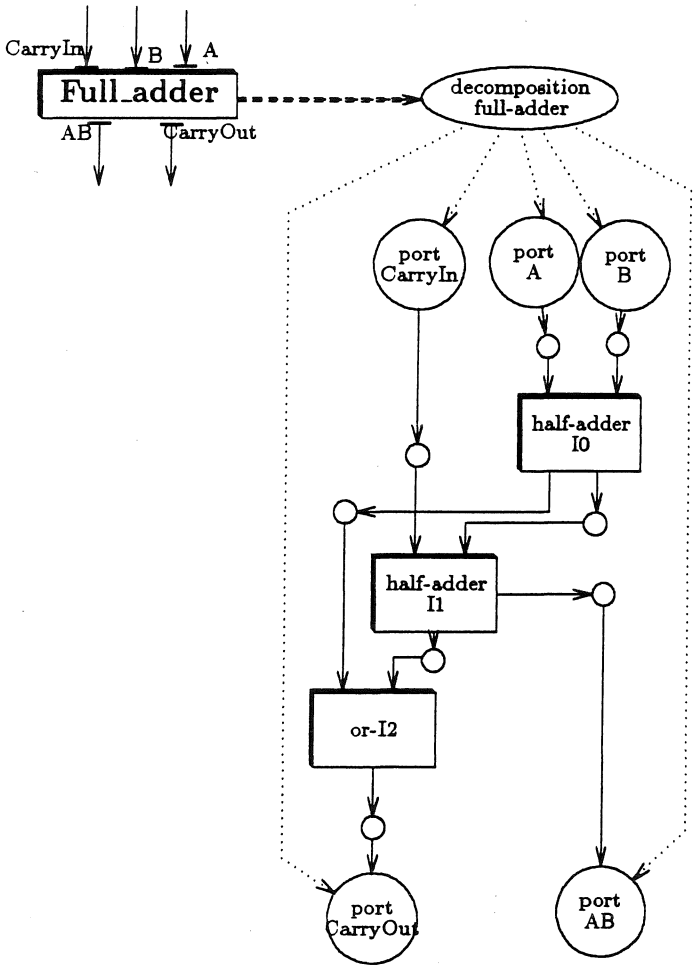


Figure 56: Annotated Component Graph for the Full-Adder Example

how this hierarchical netlist is represented internally. The five pins of the Full-Adder component, called CarryIn, A, B, CarryOut, and AB, are represented as independent port objects in the structure graph 56.

7 OUR APPROACH TOWARDS THE LINKAGE PROBLEM

7.1 Linkage between the Behavioral and the Structural Domain

The constructs in a data flow graph are implemented by one or more constructs from the structural domain, whereas the control flow constructs have no direct structural equivalent. A control flow sequencing arc for instance does not represent a real physical connection. Rather, the control flow graph models the sequencing of the behavior over time, and thus is synthesized into control logic.

The correspondence between the data flow graph and the annotated component graph is as follows. A behavioral operator or a variable access in the data flow graph gets mapped to a functional unit, register, or bus in the data path. Similarly, the data flow edges correspond to actual connections in the annotated component graph along which the values travel. Thus, data flow edges are mapped to one wire or a sequence of wires and components in the data path. This component mapping information is maintained in the form of structural annotations to the data flow graph rather than behavioral annotations to the annotated component graph. We choose this approach since the component mapping is multiplexed in time. The same hardware unit is reused multiple times. In fact, a hardware component may be bound to several data flow nodes in the same state, when scheduling across conditional branches is performed. Depending on the evaluation of the conditional branch, one of the bindings will be selected by the control unit during execution. Furthermore, the relationship between the behavioral domain and the structural domain is a many-to-many mapping for the following reason: Sequences of operations may be implementable by one functional unit, and vice versa, several units may be needed to implement one complex operation.

The designer may specify a partial design where some of the data flow operations are already bound to components. This linkage then corresponds to an externally imposed constraint for the design, that the synthesis system should observe. Hence, the links between the behavioral and the structural description have an attribute that distinguishes between a synthesized linkage (which can be modified by design tools as desired) and a fixed linkage.

7.2 A Complete Example: The Programmable Counter

Figure 57 shows a behavioral VHDL description of a 4 bit programmable up and down counter. For this example we assume a state assignment as shown in the state table in Figure 58.

```
entity counter is
 port (countin: in BIT(3 downto 0);
       up: in BIT;    - UP==1 if count up and UP==0 if count down
       count: in BIT; - COUNT==1 if count and COUNT==0 if program
       countout: out BIT(3 downto 0)
       )
end counter;
 architecture counterbody of counter is
signal I: BIT(3 downto 0);
begin
 process begin
    countout <= I;
    if (count = 1)
    then if (up = 1)
         then I <= I + 1;
         else I <= I - 1;
         end if;
    else I <= countin;
    end if;
   wait for 12ns;
 end process;
end counterbody;
```

Figure 57: VHDL Specification of a 4 bit programmable up and down counter

Then, the behavioral description of that counter is compiled into the CDFG graph depicted in Figure 59. In this figure, we draw the data flow graphs associated with each statement-block node within the control node rather than associating them by hierarchy arcs. For simplicity, all data flow nodes are depicted by circles.

Figure 60 shows one possible structural implementation of the counter (a data path). The chosen counter component has four ports, called up, down, countin, and countout. These ports represent the communication between the environment and the counter, and therefore they are visible from the outside the counter and from within.

present state	condition	(value)	actions	next state
0	-	TRUE	countout = I	1
1	count=1	TRUE	-	2
		FALSE	I = countin	0
2	up=1	TRUE	I = I + 1	0
		FALSE	I = I - 1	

Figure 58: The State Table

Each component is connected to other components via nets that link the output port of one component to the input port of another component. A net is represented by a set of connection arcs and an interconnection node with the corresponding net name. Nets that connect the control unit with the data path components are of type control.

Next, we discuss the structural information that is attached to the behavioral description in the form of annotations (see Figure 59). Note that only the data flow constructs have associated structural data. The control flow constructs model the sequencing of the behavior over time, and thus have no structural equivalent.

Each operation and variable node of the behavioral description has associated the corresponding structural component that implements it. In Figure 59, these component mapping annotations are represented by dashed lines. In particular, a data flow node is annotated by the component name, the chosen function, and the set of inputs ordered from left to right. For example, the plus operation in state S2 is bound to the ALU1 component. The selected function is ADD and its inputs from left to right are ONE and IREG. The control selection variable is described in terms of a tuple that gives the control line name (net of control type) and its associated value. For a control selection variable which is not used, a value of zero is assumed. For the previous example, the control selection variable is the tuple "CADD=1" where CADD corresponds to the control line and "1" is the value that the control unit will assign to this control line.

Similarly, a data flow net is annotated with the structural constructs by which it is implemented. This can either be a simple structural interconnection node or a sequence of interconnection nodes and component nodes. In the later case, the data flow arc also stores the function, inputs, and control selection variable of each component in the list.

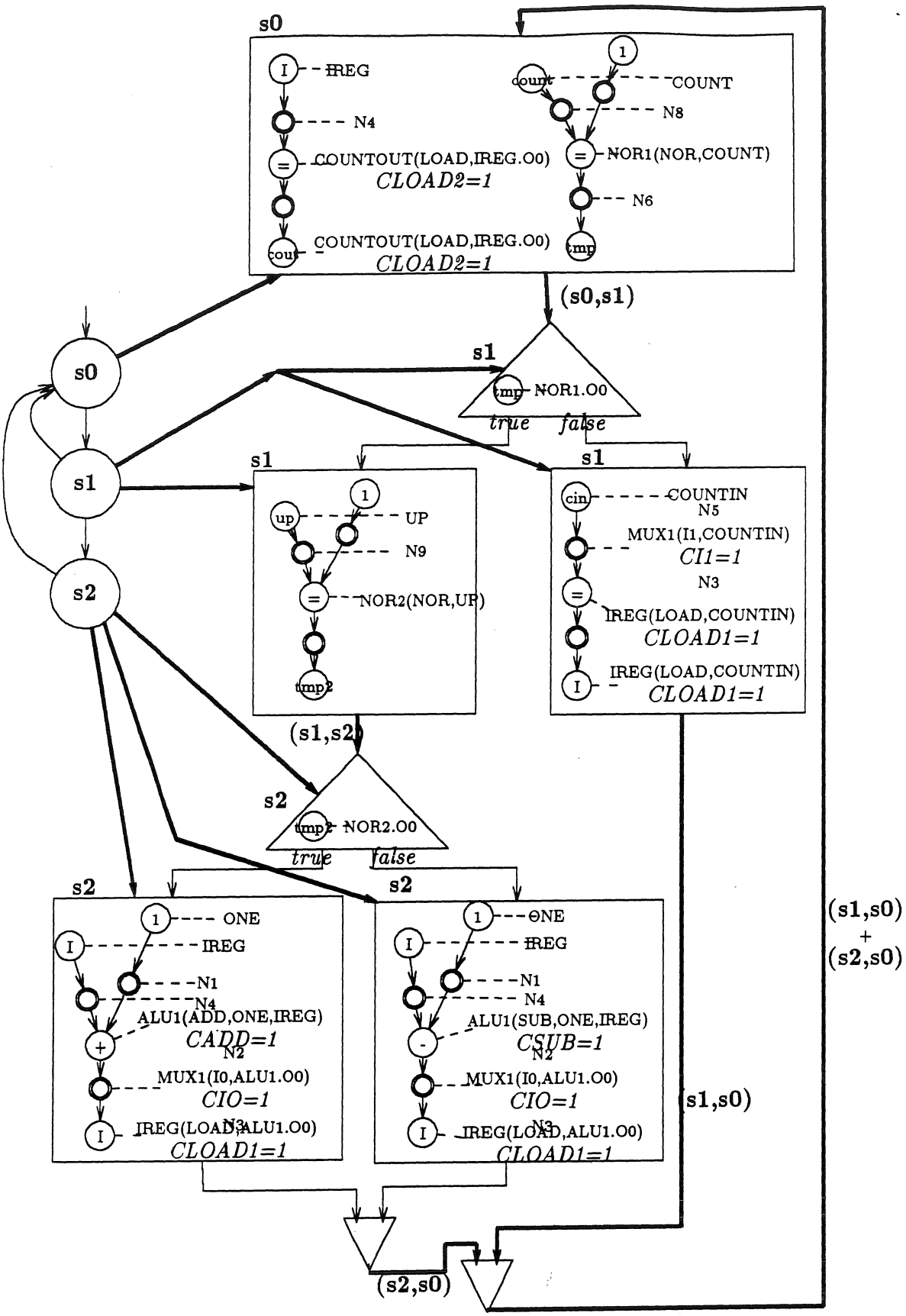


Figure 59: Complete Flow Graph Representation of the Counter

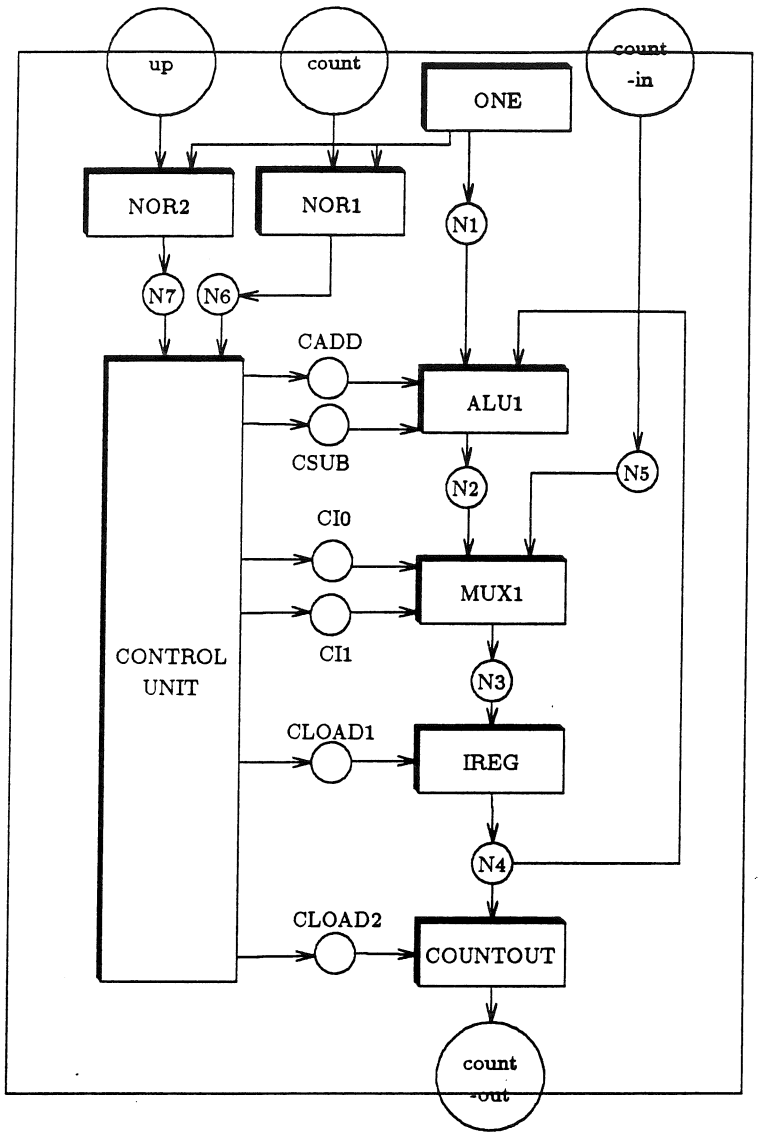


Figure 60: Data Path of the Counter Example

If a component performs only one function then a control selection is not needed. Therefore, in this case control selection information is not associated with the corresponding data flow construct. The flow graph representation given in Figure 59 is complete in as much as the structural information can be regenerated from it.

7.3 Summary of our Linkage Approach

Our approach towards the design linking problem is summarized below. It is also depicted in Figure 61. The symbols used in Figure 61 have the following meaning. A bold arrow represents explicitly maintained links. A simple arrow indicates that the links are kept implicitly, i.e., in the form of annotations. The objects at the start of the simple arrow reference the objects pointed to by the arrow. A dashed box stands for constructs that are not being maintained as separate entities in our model. Now we summarize our approach by describing how each of the possible links introduced in Section 2.4 (in particular, in Figure 1) are handled in our model.

- Connections from control flow to data flow constructs are an integral part of the proposed Control/Data Flow Graph model. A data flow graph is associated with most control flow nodes.
- Connections from each state to its associated control flow constructs is maintained as detailed in Section 5. Each state node points to one or more control flow nodes.
- Connections from a state of the control automata to the data flow constructs it contains are not maintained explicitly. The data flow constructs are however annotated by state information.
- Connections from behavioral constructs in the data flow graph to structural units in the data path graph are maintained by structural annotations to the flow graph. This component mapping information is multiplexed in time; therefore, each data flow construct is annotated by its corresponding structural unit rather than vice versa.
- Connections from each state to the data path units that are being executed in the state are not stored. They can again easily be derived by following the links from a state to its associated data flow nodes which then are annotated by the corresponding structural units. For control synthesis, not only the units used in each state but also the selected functionality are needed. A static view of these links is kept in the data path, since they correspond to the control lines from the control unit component to the data path components. A dynamic view of

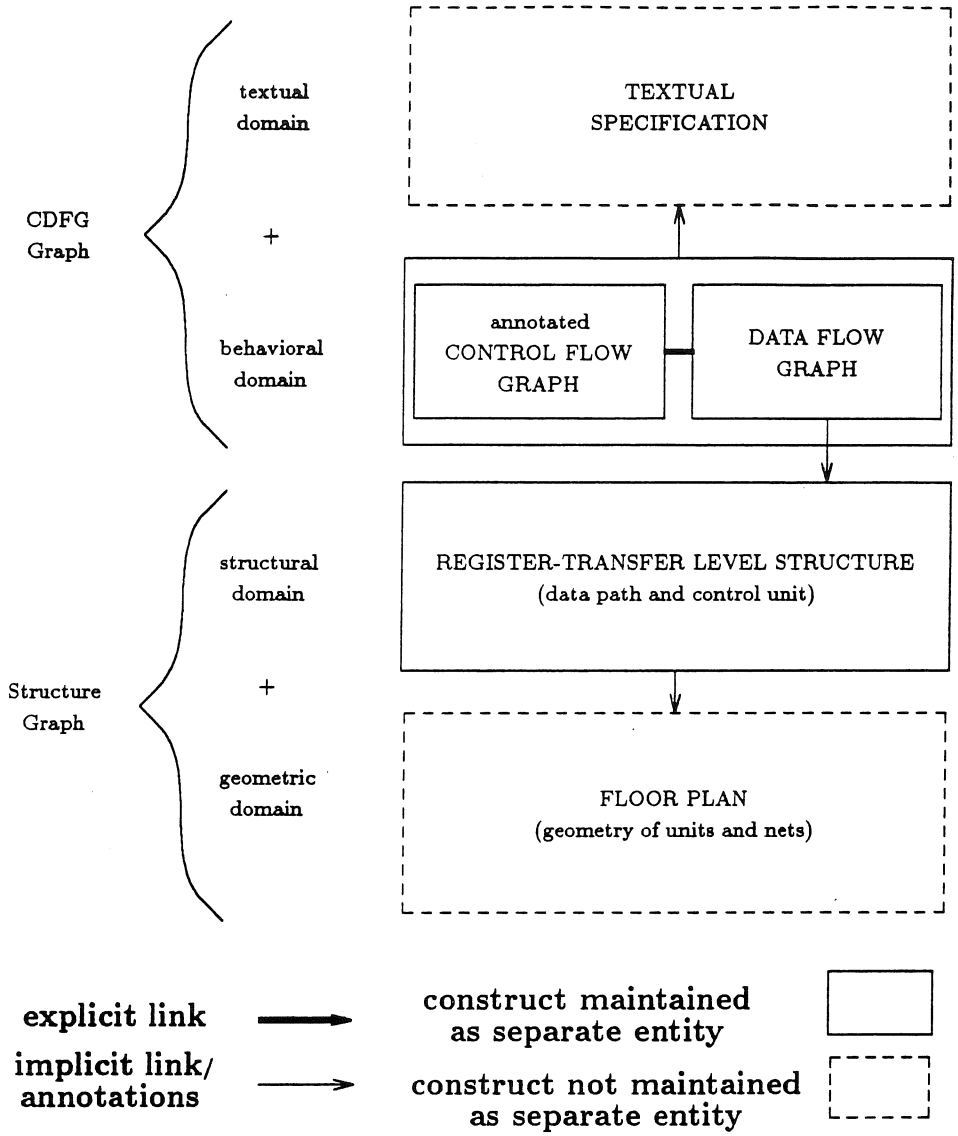


Figure 61: DDM's Approach towards the Linkage Problem

these links is expressed in the control unit description, since the selection of values for control lines per state represents the behavior of the control unit.

- Connections from the structural to the geometric domain to show the floorplan of the structure are expressed by annotating the data path components with geometric information. This mapping is not time-varying.

8 CONCLUSION

In this paper, we have presented a a unified design representation model for system and behavioral synthesis tools. The design data used by computer-aided design tools can be classified into three separate graph models: the conceptual model, the behavioral model and the structural model. The proposed design representation model, called DDM, supports all three. We have developed an Augmented Control/Data Flow Graph (CDFG) model for the behavioral model and an Annotated Component Graph (ACG) for the capture of the data path and the geometric information. The Design Entity Graph, which represents the conceptual model, will be described in a forthcoming report. Throughout this report, we give numerous examples that show how VHDL specifications can be represented by this design representation.

The proposed design representation model is a powerful vehicle for the development of consistency routines. Such routines could check for the *completeness* and the *consistency* of the design. We can check for the completeness of a design by, for instance, checking whether each operator node is bound to some structure, whether each structural unit is bound to some physical counterpart, and whether each operator node has been assigned to a state. Consistency checks that could be performed are of the following type. We can check whether a given data flow graph is bit-width consistent, i.e., if there would be any tangling bits if the graph were mapped into hardware. Other potential inconsistency problems are if two values are bound to a structural carrier at the same time (in the same state), or if there a connected graph for each value from value creation to all its uses does not exist in the component graph (ACG).

References

[1] Armstrong, J., *Chip Level Modeling with VHDL*, Prentice-Hall, 1989.

[2] Batory, D. S., and Kim, W., Modeling Concepts for VLSI CAD Objects, *ACM Tran. on Database Systems*, vol. 10, no. 3, Sep. 1985, 322 - 346.

[3] R. L. Blackburn, D. E. Thomas, and P.M. Koenig, CORAL II: linking behavior and structure in an IC design system. In *Proc. of the 25th Design Automation Conf.,* IEEE, 1988.

[4] Camposano, R. and R. M. Tabet, Design Representation for the Synthesis of Behavioral VHDL Models, Research Report, IBM General Technology Division, Burlington, VT, RC 14282, Dec. 1988.

[5] Camposano, R. and Kunzmann, A., Considering Timing Constraints in Synthesis from a Behavioral Description, *ICCD* 1986.

[6] Camposano, R. and Weber, R., Compilation and Internal Representation of Digital Systems Specification in DSL, *Sixth Conf. Latino-Americana in Informatics*, Brazil, ANAIS, Vol. II, July 85.

[7] Chen, D. and Gajski, D., An Intelligent Component Database for Behavioral Synthesis, Tech. Report 89-39, Nov. 1989.

[8] Dutt, N., A Framework for Behavioral Synthesis from Partial Design Structures, *Ph. D. Dissertation*, Uni. of Illinois, Urbana-Champaign, Jan. 1989.

[9] Dutt, N., Hadley, T. and Gajski, D., BIF: a Behavioral Intermediate Format for High Level Synthesis, Technical Report 89-03, University of California, Irvine.

[10] Gajski, D. D., and Kuhn, R., Guest Editors' Introduction: New VLSI Tools. *IEEE Computer*, vol. 16, no. 12, 11–14, Dec. 1983.

[11] Gupta, R., Cheng, W. H., Gupta R., Hardonag, I. and Breuer, M. A.. An Object-Oriented VLSI CAD Framework, *IEEE Computer*, vol. 22, no. 5, 28 – 37, May 1989.

[12] Katz, R. H., Information Management for Engineering Design, Surveys in Computer Science, 1985.

[13] Knapp, D. W., and A. C. Parker, A unified representation for design information, In *Proc. CHDL-85,* Elsvier, 1985.

[14] Kowalski, T. J., and Thomas, D. E., The VLSI Design Automation Assistant: What's in a Knowledge Base, *22nd Design Automation Conference*, 252 – 258, June 1985.

[15] Kurdahi, K. and Park, Y., Personal Communication, 1990.

[16] Lis, J. S., and D. D. Gajski, VHDL Design Representation in the VHDL Synthesis System (VSS), Tech. Rep. #89-15, Uni. of California, Irvine, 1989.

[17] Lis, J. S., and D. D. Gajski, Synthesis from VHDL, *Proc. of ICCD*, 1988, Port Chester, New York, Oct. 88.

[18] Lis, J. S., and N. Dutt, Flow graph data structures, UCI CADLAB Internal Report, July 1989.

[19] McFarland, M., The Value Trace: A Database for Automated Digital Design, Tech. Report DRC-01-04-80, Carnegie-Mellon University, Dec. 1978.

[20] Orailoglu, A. and D. D. Gajski, Flow graph representation, *Proc. of 23rd Design Automation Conf.*, Las Vegas, Jun. 1986, 503 – 509.

[21] Padua, D. A., and Wolfe, M. J., Advanced Compiler Optimizations for Supercomputers, in *Comm. of the ACM*, Dec. 1986, Vol. 29, No. 12, pg. 1184 – 1201.

[22] Ramachandran, L., Modeling and Representing Timing Information for Synthesis, UCI CADLAB Internal report, September 1990.

[23] Rundensteiner, E. A., and Bic, L., Set Operations in Data Modeling, *International Conference on Extending Data Base Technology,* March 26–30, 1990, Fondazione Cini, Venice, Italy.

[24] Rundensteiner, E. A., The Component Synthesis Algorithm. User's Manual. UCI CADLAB Internal report, May 1990.

[25] Rundensteiner, E. A., Gajski, D., and Bic, L., Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions, *ICCAD*, Nov. 1990.

[26] Rundensteiner, E. A., Flowgraph Representations for Synthesis, June 1989, unpublished.

[27] Temme, K.-H., Chip-Architekturplanung nach dem Resonanzverfahren, Ein wissensbasierter Ansatz zur Synthese algorithmischer Verfahrensbeschreibungen, (Dissertation), Universitaet Dortmund, Fachbereich Informatik, Forschungbericht Nr. 324, Nov. 1989.

[28] VHDL Language Reference Manual, Addison Wesley, 1988.

[29] Walker, R. A., and Thomas, D. E., A Model of Design Representation and Synthesis, *DAC'85*, 1985, 453–459.