Technical Report No. SNU-EE-TR-2002-8

CDFG Toolkit User's Guide

Jinhwan Jeon
Yongjin Ahn
Kiyoung Choi

August 2002

School of Electronic Engineering
Seoul National University

Haedong Digital Library
Room 312, Building 301, Seoul National University
Shinlim-dong, Kwanak-gu, Seoul 151-742, Korea
Tel: (02)880-1787 Fax: (02)882-4656
http://haedong.snu.ac.kr

Abstract

This report provides a user's guide to CDFG Toolkit, which has been developed to support fast and easy generation and manipulation of CDFG (Control/Data Flow Graph) mainly for HLS (High Level Synthesis). CDFG is generally used as an input format to an HLS system. The CDFG toolkit includes CDFG generator, CDFG to C (VHDL) converter, CDFG parser, and CDFG viewer. The CDFG generator takes VHDL intermediate format (IF) and transforms it into a textual CDFG file. The VHDL intermediate format is generated by compiling an input VHDL code using VHDL analyzer (***van***). Therefore, to obtain CDFG, you must first compile a VHDL code with van before running CDFG generator. The CDFG to C (VHDL) converter generates C (VHLD) file from a CDFG file. The CDFG parser is used for parsing a textual form of CDFG. Its C++ source codes are also provided to help you to develop your own applications using the CDFG. We developed the parser based on the OOP (Object Oriented Programming) concept for the extendibility and modularity of programming. Finally, the CDFG viewer provides a GUI (Graphic User Interface) for the CDFG. It shows the CDFG graphically to the user. It also shows information related with HLS, including scheduling and binding.

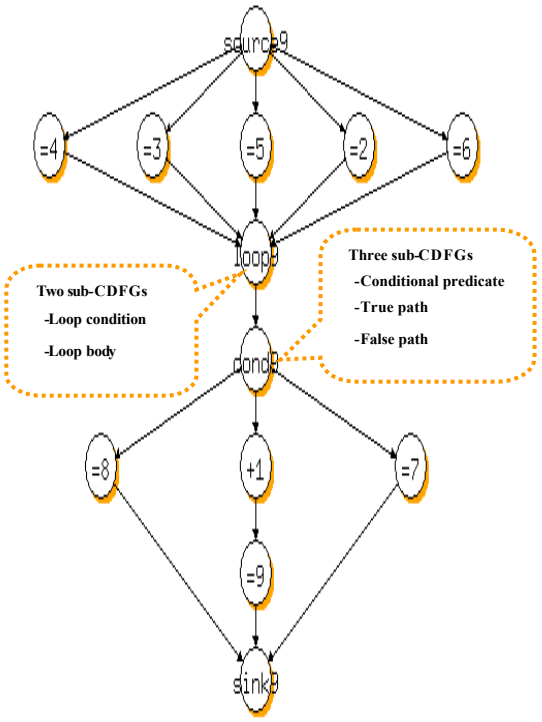
Table of Contents

1. Introduction

The CDFG Toolkit provides convenient tools to the users so that they can speed up the development of various tools for systems design. It includes CDFG generator, CDFG to C (VHDL) converter, CDFG parser, and CDFG viewer. It has originally been developed to support fast and easy generation and manipulation of CDFG (Control/Data Flow Graph) mainly for HLS (High Level Synthesis)[2].

CDFG is generally used as an input format to an HLS system. The detailed structure and information contained in it differ from system to system. Figure 1.1 shows the structure of the CDFG that we use in our toolkit. Figure 1.1 (a) is an example of VHDL description and (b) is the corresponding CDFG.

```
entity example is
  port (Xinport: in integer;
        Xoutport: out integer;
        DXport: in integer;
        Aport: in integer;
        Yinport: in integer;
        Youtport: out integer;
        Uinport: in integer;
        Uoutport: out integer);
end example;
architecture example of example is
begin
P1 : process (Aport, DXport, Xinport, Yinport, Uinport)
  variable x_var,y_var,u_var, a_var, dx_var, tmp_var: integer ;
  variable x1, y1, t1,t2,t3,t4,t5,t6: integer ;
begin
 x_var := Xinport;  a_var := Aport; dx_var := DXport;
 y_var := Yinport; u_var := Uinport;
 while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    y1 := u_var * dx_var;
    u_var := t6 - t5;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
 if (x_var = y_var) then
            tmp_var := y_var + x_var;
            x_var := tmp_var;
 else
            tmp_var := x_var * y_var;
            y_var := tmp_var;
 end if;
 Xoutport <= x_var;
 Youtport <= y_var;
 t1 := x_var + y_var;
 Uoutport <= t1;
end process P1;
end example;
configuration cfg_EXAMPLE of EXAMPLE is
   for example
   end for;
end cfg_EXAMPLE;
```

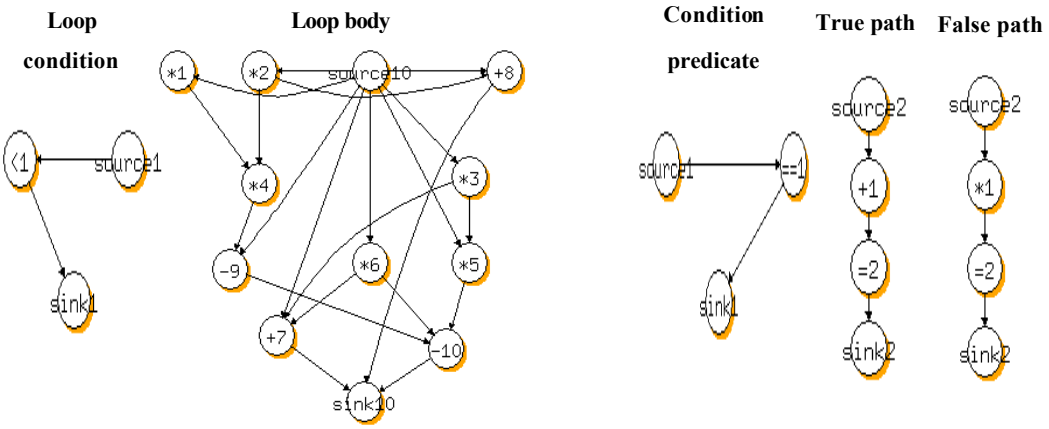


(a) VHDL description **(b) CDFG**

Figure 1. 1. An example of the CDFG.

The CDFG is basically an acyclic graph with nodes (vertices) and edges. It is in the form of combining two graphs: data-flow graph including operations and data dependency and control-flow graph including conditional branching, iteration, and module. The CDFG has hierarchical structure where all the sub-graphs are described in the same form. Each sub-graph is a polar graph containing a source node and a

sink node that model no operation. We classify a node into two types, namely, operation node and hierarchical node. Figure 1.1 also shows the hierarchy of the CDFG. The nodes,'***loop9'***and ***'cond9'*** are hierarchical nodes. There are three types of hierarchical node: module node, condition node, and iteration node. The module node models top entity, process, entity instantiation, and procedure (function) call in VHDL. The condition node models conditional construct. If the condition node represents *if.. then. else..* construct, there exist three child sub-graphs, that is, sub-CDFGs which correspond to conditional predicate, true path, and false path, respectively. Figure 1.2 (a) shows the CDFG for the iteration node (***loop9***) in Figure 1.1 (b). If the condition node represents *case* construct, there exist a sub-graph for conditional predicate and one or more sub-graphs where each sub-graph corresponds to each *when* construct in VHDL. The iteration node corresponds to the loop construct of VHDL. In the iteration node, there are two child sub-graphs which correspond to loop condition and loop body, respectively. Figure 1.2 (b) shows the CDFG for the condition node (***cond9***) in Figure 1.1 (b).



(a) CDFG for the iteration node **(b) CDFG for the condition node**

Figure 1. 2. Sub-CDFGs for hierarchical nodes.

Edges that connect vertices represent data dependency and control dependency. The control dependency represents control signals transferred from the conditional predicate nodes (usually comparison operations). A control dependency edge exists only in a conditional sub-graph, which belongs to a condition node or an iteration node. See Chapter 2 and Chapter 4 for details of the CDFG.

This report is structured as follows. The next chapter presents the CDFG generator which transforms a VHDL intermediate format (IF) file into a CDFG file. Chapter 3 explains the CDFG to C and CDFG to VHDL translator. Chapter 4 explains the CDFG parser, which builds a graph from a CDFG file generated by our CDFG generator. Finally, Chapter 5 explains the CDFG viewer and shows the usage of it.

2. CDFG Generator

The CDFG generator (***cdfggen***) transforms intermediate format (IF) file into textual control data flow graph (CDFG) file. The intermediate format file is generated by compiling an input VHDL code using VHDL analyzer, van [1]. Therefore, you must compile the VHDL code with van before running CDFG generator.

2.1 Usage

The usage of cdfggen is as follows.

Usage: ***cdfggen*** [options] top_configuration_name

Options:

-v : enable verbose mode

-P : expand all the packages used in the source VHDL code into CDFG file

-o *output_file* : set output file name

-C *format* : set output format (0, 1, 2). '2' is the default format for which we provide C++ parser.

-D: enable debug mode

-nocycle : In some examples(synchronous circuit), there may exist cycle in the graph. If this option is on, all the cycles are removed in the graph. (Generally not used.)

-noconv : If the same entity is instantiated multiple times, we convert name such that each instance has the unique name. If this option is on, we do not convert name. (Generally not used.)

-nosplitprocess : In synchronous circuit, we need to split a process into multiple parts in order to represent correct dependencies. If this option is on, we don't split process for that case. (Generally not used.)

-i *indent_size* : set the size of indent.

-f *vhdlfile* : Generally you should compile *vhdlfile* before running cdfggen. If this option is on cdfggen executes van before generating CDFG. It has the same effect as executing van and cdfggen in the following sequence.

van *vhdlfile*.vhd; cdfggen cfg_*vhdfile*

-s *clock_signal* : set clock signal name for synchronous circuit.

Though ***cdfggen*** reads IF file generated by van. We don't need to remember all the names of IF files to

make a CDFG file. We have only to give the top configuration name of the design to ***cdfggen***. Therefore, the input VHDL file should be described in complete form containing configuration construct. Figure 2.1 shows an example VHDL code which describes 11th order FIR filter.

```
-----------------------------------------------------------------------------------------
-- Example VHDL file describing an 11 th order FIR filter
-----------------------------------------------------------------------------------------
entity fir11 is
port (
    inp0: in real;
    outp: out real);
end fir11;

architecture fir11_beh of fir11 is
begin

process (inp0)
    variable inp1, inp2, inp3, inp4, inp5, inp6, inp7, inp8, inp9, inp10: real;
    variable acc0, acc1, acc2, acc3, acc4,acc5, acc6, acc7, acc8, acc9: real;
begin
    acc0 := inp10 * (-0.001953125);
    acc1 := inp9 * (0.003906250) + acc0;
    acc2 := inp8 * (-0.007812500) + acc1;
    acc3 := inp7 * (0.019531250) + acc2;
    acc4 := inp6 * (-0.066406250) + acc3;
    acc5 := inp5 * (0.750000000) + acc4;
    acc6 := inp4 * (-0.066406250) + acc5;
    acc7 := inp3 * (0.019531250)+ acc6;
    acc8 := inp2 * (-0.007812500) + acc7;
    acc9 := inp1 * (0.003906250) + acc8;
    outp <= inp0 * (-0.001953125) + acc9;

    inp10 := inp9;
    inp9 := inp8;
    inp8 := inp7;
    inp7 := inp6;
    inp6 := inp5;
    inp5 := inp4;
    inp4 := inp3;
    inp3 := inp2;
    inp2 := inp1;
    inp1 := inp0;
end process;
end fir11_beh;

configuration cfg_fir11 of fir11 is
    for fir11_beh
    end for;
end cfg_fir11;
```

Figure 2.1. Example VHDL code for 11th order FIR filter.

The example is described in complete form containing entity, architecture body and configuration constructs. To generate a CDFG file named 'fir11.cdfg' we have only to run cdfggen as follows.

cdfggen –o fir11.cdfg cfg_fir11

Note that we don't give the VHDL file name but the top configuration name to *cdfggen*. You should compile the VHDL file before running *cdfggen* by running van as follows.

van fir11.vhd

Figure 2.2 shows the CDFG file generated by cdfggen for the FIR filter example.

```
*****************************************************
* CDFG for FIR11 generated by 'cdfggen'
*
* Date: Mon Jun  7 15:23:43 1999
* User: jeonjinh
*****************************************************
edge 0 1 INP0 REAL 1 32
edge 1 -1 OUTP REAL 1 32
node 0 source - - -
node 1 FIR11 - - -
  length_type INP0 REAL 32 in - to -
  length_type OUTP REAL 32 out - to -
   subgraph  1
     edge  0  1  INP0  REAL  1  32
     edge  0  1  INP0  event  1  1
      edge  1  -1  OUTP  REAL  1  32
     node  1  mod_1 - - - (PROCESS)
        length_type INP1 REAL 32 - - to -
        length_type INP2 REAL 32 - - to -
        length_type INP3 REAL 32 - - to -
        length_type INP4 REAL 32 - - to -
        length_type INP5 REAL 32 - - to -
        length_type INP6 REAL 32 - - to -
        length_type INP7 REAL 32 - - to -
          length_type INP8 REAL 32 - - to -
        length_type INP9 REAL 32 - - to -
       length_type INP10 REAL 32 - - to -
        length_type ACC0 REAL 32 - - to -
        length_type ACC1 REAL 32 - - to -
        length_type ACC2 REAL 32 - - to -
         length_type  ACC3 REAL 32 - - to -
        length_type ACC4 REAL 32 - - to -
        length_type ACC5 REAL 32 - - to -
        length_type ACC6 REAL 32 - - to -
        length_type ACC7 REAL 32 - - to -
        length_type ACC8 REAL 32 - - to -
        length_type ACC9 REAL 32 - - to -
        length_type  _tmp_0_4ACC1 REAL 32 - - to -
        length_type  _tmp_0_6ACC2 REAL 32 - - to -
        length_type  _tmp_0_8ACC3 REAL 32 - - to -
        length_type  _tmp_0_10ACC4 REAL 32 - - to -
        length_type  _tmp_0_12ACC5 REAL 32 - - to -
          length_type  _tmp_0_14ACC6 REAL 32 - - to -
        length_type  _tmp_0_16ACC7 REAL 32 - - to -
        length_type  _tmp_0_18ACC8 REAL 32 - - to -
        length_type  _tmp_0_20ACC9 REAL 32 - - to -
```

```
length_type _tmp_0_22OUTP REAL 32 - - to -
 subgraph 1
   edge 0 1 INP10 REAL 1 32
  edge 0 1 -0.001953125 REAL 1 32
 edge 0 2 INP9 REAL 1 32
 edge 0 2 0.003906250 REAL 1 32
 edge 2 12 _tmp_0_4ACC1 REAL 1 32
 edge 1 12 ACC0 REAL 1 32
 edge 0 3 INP8 REAL 1 32
    edge 0 3 -0.007812500 REAL 1 32
 edge 3 13 _tmp_0_6ACC2 REAL 1 32
 edge 12 13 ACC1 REAL 1 32
 edge 0 4 INP7 REAL 1 32
 edge 0 4 0.019531250 REAL 1 32
 edge 4 14 _tmp_0_8ACC3 REAL 1 32
 edge 13 14 ACC2 REAL 1 32
   edge 0 5 INP6 REAL 1 32
  edge 0 5 -0.066406250 REAL 1 32
 edge 5 15 _tmp_0_10ACC4 REAL 1 32
 edge 14 15 ACC3 REAL 1 32
 edge 0 6 INP5 REAL 1 32
 edge 0 6 0.750000000 REAL 1 32
 edge 6 16 _tmp_0_12ACC5 REAL 1 32
 edge 15 16 ACC4 REAL 1 32
 edge 0 7 INP4 REAL 1 32
  edge 0 7 -0.066406250 REAL 1 32
 edge 7 17 _tmp_0_14ACC6 REAL 1 32
 edge 16 17 ACC5 REAL 1 32
 edge 0 8 INP3 REAL 1 32
 edge 0 8 0.019531250 REAL 1 32
  edge 8 18 _tmp_0_16ACC7 REAL 1 32
 edge 17 18 ACC6 REAL 1 32
 edge 0 9 INP2 REAL 1 32
  edge 0 9 -0.007812500 REAL 1 32
 edge 9 19 _tmp_0_18ACC8 REAL 1 32
 edge 18 19 ACC7 REAL 1 32
 edge 0 10 INP1 REAL 1 32
  edge 0 10 0.003906250 REAL 1 32
 edge 10 20 _tmp_0_20ACC9 REAL 1 32
 edge 19 20 ACC8 REAL 1 32
 edge 0 11 INP0 REAL 1 32
  edge 0 11 -0.001953125 REAL 1 32
 edge 11 21 _tmp_0_22OUTP REAL 1 32
  edge 20 21 ACC9 REAL 1 32
 edge 0 22 INP9 REAL 1 32
 edge 1 22 depend bool 1 1 (anti)
 edge 0 23 INP8 REAL 1 32
 edge 22 23 depend bool 1 1 (anti)
 edge 2 23 depend bool 1 1 (anti)
 edge 0 24 INP7 REAL 1 32
  edge 23 24 depend bool 1 1 (anti)
 edge 3 24 depend bool 1 1 (anti)
 edge 0 25 INP6 REAL 1 32
 edge 24 25 depend bool 1 1 (anti)
 edge 4 25 depend bool 1 1 (anti)
 edge 0 26 INP5 REAL 1 32
 edge 25 26 depend bool 1 1 (anti)
   edge 5 26 depend bool 1 1 (anti)
 edge 0 27 INP4 REAL 1 32
 edge 26 27 depend bool 1 1 (anti)
 edge 6 27 depend bool 1 1 (anti)
 edge 0 28 INP3 REAL 1 32
 edge 27 28 depend bool 1 1 (anti)
 edge 7 28 depend bool 1 1 (anti)
```

```
edge 0 29 INP2 REAL 1 32
edge 28 29 depend bool 1 1 (anti)
edge 8 29 depend bool 1 1 (anti)
edge 0 30 INP1 REAL 1 32
edge 29 30 depend bool 1 1 (anti)
edge 9 30 depend bool 1 1 (anti)
edge 0 31 INP0 REAL 1 32
edge 30 31 depend bool 1 1 (anti)
edge 10 31 depend bool 1 1 (anti)
edge 31 -1 INP1 REAL 1 32
edge 30 -1 INP2 REAL 1 32
edge 29 -1 INP3 REAL 1 32
edge 28 -1 INP4 REAL 1 32
edge 27 -1 INP5 REAL 1 32
edge 26 -1 INP6 REAL 1 32
edge 25 -1 INP7 REAL 1 32
edge 24 -1 INP8 REAL 1 32
edge 23 -1 INP9 REAL 1 32
edge 22 -1 INP10 REAL 1 32
edge 21 -1 OUTP REAL 1 32
node 1 * - - -
node 2 * - - -
node 3 * - - -
node 4 * - - -
node 5 * - - -
node 6 * - - -
node 7 * - - -
node 8 * - - -
node 9 * - - -
node 10 * - - -
node 11 * - - -
node 12 + - - -
node 13 + - - -
node 14 + - - -
node 15 + - - -
node 16 + - - -
node 17 + - - -
node 18 + - - -
node 19 + - - -
node 20 + - - -
node 21 + - - -
node 22 = - - -
node 23 = - - -
node 24 = - - -
node 25 = - - -
node 26 = - - -
node 27 = - - -
node 28 = - - -
node 29 = - - -
node 30 = - - -
node 31 = - - -
node -1 sink - - -
node 0 source - - -
end
node -1 sink - - -
node 0 source - - -
end
node -1 sink - - -
```

Figure 2.2. CDFG file generated by *cdfggen*

If we describe the VHDL file named '*test*.vhd' such that the top configuration name is cfg_*test*, we can generate a CDFG file directly from a VHDL file by running ***cdfggen*** as follows.

cdfggen –f test.vhd

In this case, *cdfggen* calls *van* before generating a CDFG file.

2.2 CDFG Format

Figure 2.3 shows the BNF definition of our CDFG format.

```
CDFG ::=
        edge_def_list | node_def_list [CDFG]
edge_def_list ::= edge_def [edge_def_list]
node_def_list ::= node_def [node_def_list]
edge_def ::=
        edge predecessor_id successor_id name type weight bits [attribute_def_list]
node_def ::=
        {module_def | operator_def | condition_def | iteration_def} [attribute_def_list]
operator_def ::=
        node id oper_type hw_speed sw_speed hw_sw_type
oper_type ::=
        "=" | "&" | "|" | "nand" | "nor" | "xor" | "==" | "!=" | "<" | "<=" | ">" | ">="
        "+" | "-" | "*" | "/" | "%" | "concat" | "abs" | "exp" | "~"
condition_def ::=
        node id cond hw_speed sw_speed hw_sw_type
                condition id
                        subgraph_body_def
                end
                if_condition_body_def | case_condition_body_def
if_condition_body_def ::=
        true id
                subgraph_body_def
        end
        [false id
                subgraph_body_def
        end]
case_condition_body_def ::=
        subgraph case_choice
                subgraph_body_def
        end
        [case_condition_body_def]
subgraph_def ::=
        subgraph id
                subgraph_body_def
        end
iteration_def ::=
        node id loop hw_speed sw_speed hw_sw_type
                iteration id
                        subgraph_body_def
                end
                subgraph_def
module_def ::=
        node id module_name hw_speed sw_speed hw_sw_type
                [length_type_def_list]
                subgraph id
                        [assign_list]
                        subgraph_body_def
                end
length_type_def ::=
        length_type name type bits type_kind_def range_def [attribute_def_list]
```

```
length_type_def_list ::=
        length_type_def [length_type_def_list]
assign_def ::=
        assign actual formal
assign_def_list ::=
        assign_def [assign_def_list]
type_kind_def ::=
        "in" | "out" | "inout" | "subtype" | "array"
range_def ::=
        range_left    {to | downto}    range_right
subgraph_body_def ::=
        edge_def_list | node_def_list [subgraph_body_def]
attribute_def_list ::=
        attribute_def [attribue_def_list]
attribute_def ::=
        ({literal_list | attribute_def} [literal_list | attribute_def] )
literal_list ::=
        literal [literal_list]
```

Figure 2.3. CDFG format.

Basically a CDFG is an acyclic graph which is composed of nodes and edges. The CDFG has hierarchical structure where all the sub-graphs are described in the same form. Each subgraph contains source node with id 0 and sink node with id −1. In each subgraph, all the edges transferring input data are connected to the source node and all the edges transferring output data are connected to the sink node. Therefore, there exists no cycle in the graph.

We classify a node into two types: operation node and hierarchical node. The operation node corresponds to each operation in the input VHDL code (such as addition, subtraction, comparison and multiplication). The hierarchical node includes module, condition and iteration node. By way of the hierarchical node we can step into different hierarchy in the graph. The module node corresponds to top entity, process, entity instantiation, and procedure (function) call in VHDL. Note that top entity *FIR11* is mapped to module node *FIR11* in Figure 2.2. The condition node corresponds to the conditional construct of VHDL. The condition node has multiple child sub-graphs based on the type of the conditional construct. If the condition node represents *if.. then.. else..* construct, there exist three child sub-graphs which correspond to conditional predicate, true path and false path, respectively. If the condition node represents *case* construct, there exist a subgraph for conditional predicate and one or more sub-graphs where each subgraph corresponds to each *when* construct in VHDL. In this case, the id of each subgraph corresponds to the condition in *when* construct. The iteration node corresponds to the *loop* construct of VHDL. In the iteration node, there are two child sub-graphs which correspond to loop condition and loop body, respectively.

Each edge in CDFG represents dependency between nodes (including RAW(Read After Write) dependency, WAR(Write After Read) dependency, WAW(Write After Write) dependency and control

dependency). If an edge represents RAW dependency, the edge corresponds to data transfer between nodes whose name is the same as variable name or temporary name generated by *cdfggen*. If the edge represents WAR or WAW dependency, the edge has a name "depend" with a dependency type ('anti' for WAR dependency and 'output' for WAW dependency). In Figure 2.2, we can see that there exist *anti* dependencies between assign operations. The type of the dependency is expressed by the *attribute field* in our CDFG format. The attribute field is defined as a sub string between '(' and ')'. We define the attribute field for the extendibility of our CDFG form. We can add any type of information by adding string to the attribute field. In CDFG, a control dependency edge exists only in a conditional subgraph which belongs to a condition node or a iteration node. The control dependency represents control signal transferred from the conditional predicate node (usually a comparison operation). In this case, the name of the control dependency edge is 'ctrl' and the type is 'bool'.

In CDFG, the symbols defined and used within subgraph is put in *length_type* field. The *length_type* field plays the role of symbol table. This field shows the name, type and size of a variable defined within subgraph. It also contains the interface signals whose type is IN, OUT or INOUT. Note that interface signal *INP0* and *OUTP* is put in *length_type* of *FIR11* node in Figure 2.2.

When we describe VHDL source code, we can use multiple instances of the same entity. In this case, each instance is mapped to a unique node (a module node) in our CDFG. If the same entity is instantiated multiple times, each instance has different I/O signals. In that case, we map the actual I/O signal to the formal I/O signal by using assign field. Such mapping is necessary because we use the same form of subgraph body even if it is multiply instantiated. In the following example, instance *I1* of component A has two formal I/O signals: *inp* and *outp*, which are mapped to actual I/O signals: *sig1* and *sig2*, respectively. In the CDFG, they are represented as *'assign sig1 inp'* and *'assign sig2 outp'*, respectively.

```
entity TOP is
end TOP;
architecture structure of TOP is
  signal sig1, sig2: integer;
  signal sig3, sig4: integer;
  component A is
    port (inp: in integer, outp: out integer);
  end component;
begin
  I1: A port map (inp=>sig1, outp=>sig2);
  I2: A port map (inp=>sig3, outp=>sig4);
end TOP;
```

2.3 Recommended VHDL description style suitable for CDFG generation

CDFG generator does not support all the VHDL description style because there are many features that cannot be represented in the form of graph in IEEE VHDL standard. Following is recommended VHDL description style such that *cdfggen* can generate CDFG file safe and sound. If you want to describe in other style or want to use any other VHDL constructs not listed here, try it. The CDFG generator may produce a CDFG file, in some cases, error message or segmentation fault, otherwise.

```
-- You can describe Library or Use statement here. They are optional.
Library IEEE;
use WORK.all;

entity A is
        port ( .... ); -- describe port list
end A;

architecture B of A is
        procedure ...           – We support procedure . but we cannot guarantee...
        end procedure;          – We still support function but we also cannot guarantee...

        component D
                ...
        end component;
begin
I1: D port map ( ...);          – You can instantiate other entity. It is represented in a module node where port mapping is
                                – represented in the form of 'assign Actual Formal' in CDFG
sig1 <= sig2 + sig3;            – You can use concurrent statements. But we recommend sequential statements within a
process.

P1: process             -- We recommend behavior is described within process.
        if ... then             – if statement is converted into a condition node. Conditional predicate is converted into
                ...             – a condition subgraph belonging to the condition node
                ...             – true path is converted into a true subgraph.
        else                    – false path is converted into a false subgraph
                if ... then     – we support nested if statements.
                end if
        end if;

        case ... is                     -- case statement is converted into a condition node.
                when C1 =>              -- each when statement corresponds each subgraph belonging to the condition node
                        ...             -- where the name of subgraph corresponds to the case that the subgraph is
                when C2 =>              -- activated.
                        ...
        end case;

        while C0 loop           – while statement is converted into a iteration node where conditional predicate C0
                ...             – corresponds to a condition subgraph belonging to the iteration node.
        end loop;

        c := a + b;             – We support any kinds of arithmetic and logical operations.
        s <= a * b;
end process
end B;

configuration cfg_A of A is     – You should describe configuration in complete form.
        for A
                for I1: D
                        use work.cfg_D;
                end for;
```

```
                end for;
end cfg_A;
```

3. CDFG to C (VHDL) Translator

3.1 CDFG to C converter

You can generate C code using ***cdfg2c***. It generates behavioral C code not only from behavioral VHDL code but also from structural VHDL code. The synchronous circuit described in structural VHDL code can be converted into behavioral C code whose behavior is described cycle by cycle. It is similar to the function of cycle-based simulator. To convert structural VHDL code into C code, you should describe the synchronous circuit in the following form.

```
architecture STRUCTURAL of TOP is
 process (clk, sig1)            -- You should describe the behavior within a process
   begin
      if (clk'event and clk=='1') then          -- You should check the clock event in this form
                                                -- You may describe the behavior between clock ticks here
      end if;
 end process;
```

The usage of the ***cdfg2c*** is as follows.

usage: ***cdfg2c*** [options] input

options:

-v : set verbose mode

-a : Assume all the nodes are in SW part. This option is valid when the CDFG is partitioned into HW part and SW part.

-D : set debug mode

-L library : set library (IEEE(default), STD)

-o output : set output file name

-i input : set input guide file. This option is valid when the input guide file is partitioned into HW part and SW part.

-g : generate co-simulation target C.

-S : generate code according to initial schedule order

-s : generate simulation code

-m : generate monitor code

-nosplit : don't split inout port. In some case we need to split inout port into input and output ports. If this options is on we don't split the port.

-noempty : don't generate empty function. If some hierarchical node has empty subgraph, we

don't need to generate code for this node.

-flat : flatten symbols

-t : split code for temporary edges.

-Oc : minimize code size

-c : genreate monitor code in each control step

-fix : treat real variables as fixed point variables.

3.2 CDFG to VHDL converter

The purpose of this tool is to confirm if the generated CDFG is equivalent to the input VHDL code. The generated code may have different code from the original code. However, they are functionally equivalent.

4. CDFG Parser

We implemented a CDFG parser using C++ under UNIX environment. We use OOP concept for the ease of usage and for the extendibility of data structure. The parser builds a graph from a CDFG file generated by our CDFG generator. Figure 4.1 shows class hierarchy of the parser. *Node* is a base class for *Condition*, *Iteration*, and *Module* classes. Each *Node* class corresponds to each ***node*** in CDFG definition (see Figure. 2.3). In the same way, *Condition*, *Iteration* and *Module* classes correspond to ***condition***, ***iteration***, and ***module***, respectively. Each *Subgraph* contains lists of edges and nodes. They are represented as List Classes which are the list of pointers to Basic Classes. The name of each Basic Class is the same as each identifier in CDFG definition (see Figure 2.3) except for the Auxiliary Classes. The auxiliary classes corresponds to *attribute* field in CDFG definition which is represented in LISP form. In the following section, we'll explain each class's member data and member functions, not all of them but a few important ones.

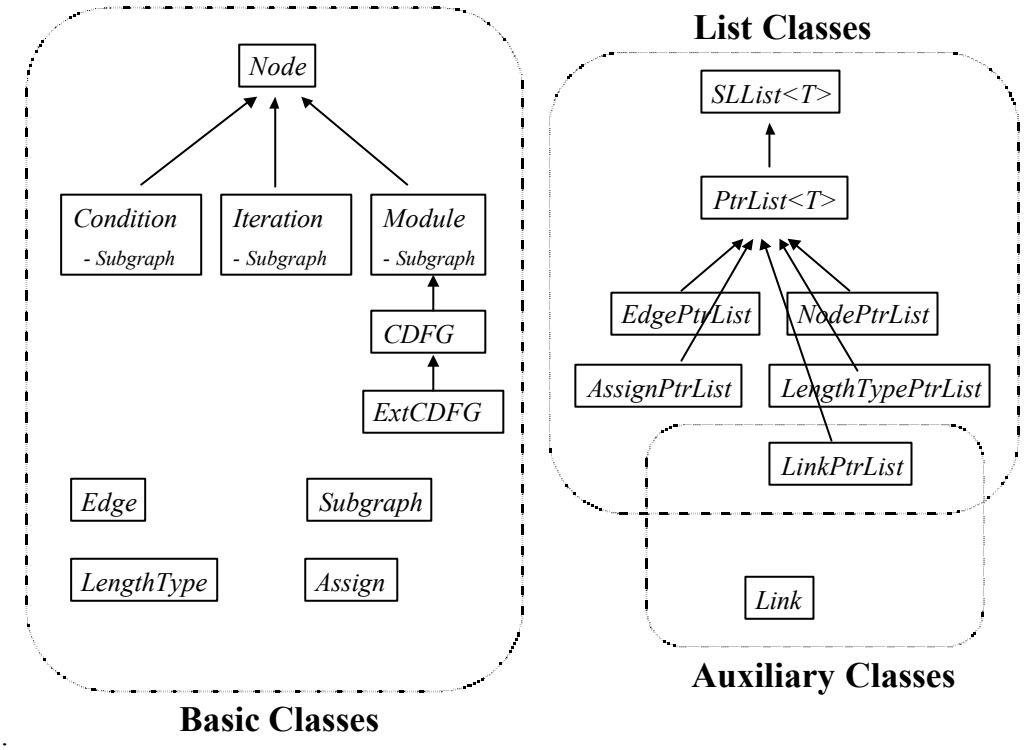


Figure 4. 1 Class hierarchy of CDFG parser.

4.1 *Node* Class

Member Data:

☞***EdgePtrList Node::preds*;**

Description:

It contains pointer to input edges. The input edges are not always data transfer edges because some edge represent control dependency or semantic meaning.

Related Functions:

EdgePtrList* Node::getPreds()

void Node::addPred(Edge* e);

void Node::subPred(Edge *e);

Edge* Node::getPred(int index, int data_flag);

☞***EdgePtrList Node::succs;***

Description:

It contains pointer to output edges. The input edges are not always data transfer edges because some edge represent control dependency or semantic meaning.

Related Functions:

EdgePtrList* Node::getSuccs();

void Node::addSucc(Edge* e);

void Node::subSucc(Edge *e);

Edge* Node::getSucc(int index, int data_flag);

☞***int Node::mark;***

Description:

This field is reserved for user. It can be used to mark nodes during graph traversal – especially during list scheduling.

Related Functions:

int Node::getMark(int m);

void Node::setMark(int m);

☞***short Node::type;***

Description:

It represent the type of a node where the types are defined as **enum N_TYPE** in cdfg.h which is defined as

```
enum N_TYPE {
        N_NULL=0,           // default
        N_OPER=1,           // operational node
        N_MOD=2,            // module
        N_ITER=4,           // iteration
```

```
        N_COND=8                    // condition
};
```

You can traverse node list in a subgraph and perform proper action for each node type as in the following example.

```
void Test(Subgraph *subg)
{
        NodePtrList &nlist = *subg->getNodes();
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                Node *n = nlist(pi);
                switch (n->getType()) {
                        case N_OPER:
                                ...
                        case N_MOD:
                                ...
                        case N_COND:
                                ...
                        case N_ITER:
                                ...
                }
        }
}
```

Related Functions:

int Node::setType(int t);

N_TYPE getType();

☞***String Node::name;***

Description:

It contains the name of a node (*this Node*). It has valid name only when *this Node* is not an operation node but *Condition*, *Iteration*, or *Module* node. The variable contains string "cond", "loop", and "{name of module}" for *Condition*, *Iteration*, and *Module*, respectively.

Related Functions:

char* Node::getName();

void Node::setName(const char *n);

☞***int Node::id;***

Description:

This field indicates the id of each node. The id has positive value, where id values of 0 and –1 are reserved for source and sink nodes, respectively.

Related Functions:

int Node::setId(int i);

int Node::getId();

☞***int Node::op;***

Description:

This field indicates the operation kind of the operation node. The kind of operation is defined in **token.h** as *operator_kind* and *my_operator_kind* enumeration types.

Related Functions:

int Node::getOp();

void Node::setOp(int op);

☞***int Node::_no;***

Description:

This field indicates the line number at which *this Node* is defined.

Related Functions:

void Node::setNo(int n);

int Node::getNo();

☞***Subgraph * Node::parent;***

***Description*:**

It represents the pointer to parent Subgraph where the parent Subgraph belongs to one of Module, Iteration, Condition, and CDFG classes.

***Related Functions*:**

Subgraph* Node::getParent();

Node* Node::getPNode();

Node* Node::getPModule();

☞***LinkPtrList Node::_attrib;***

***Description*:**

List of attributes stored in Link class. The contents of attribute is described in textual form wrapped around by parentheses, which are described as follows in the case of a node but not limited to it.

node 1 .... (cstep 1) (bind mult)

Textual description for node *Textual description for attributes*

***Related Functions*:**

LinkPtrList* Node::getAttrib();

Link* Node::findAttrib(const char *name);

LinkPtrList* Edge::getAttrib();

Link* Edge::findAttrib(const char *name);

LinkPtrList* LengthType::getAttrib();

Link* Edge::findAttrib(const char *name);

☞***int Node::tstart, tend;***

☞***int Node::pid;***

***Description*:**

These are reserved member variables for scheduling and binding.

Related Functions:

int Node::getTstart();

int Node::getTend();

void Node::setTstart(int);

void Node::setTend(int);

int Node::getPid();'

void Node::setPid(int);

☞***int Node::hw_sw_type;***

☞***int Node::sw_speed;***

☞***int Node::hw_speed;***

Description:

These are reserved member variables for scheduling and binding.

☞***void * Node::info;***

Description:

This variable indicates the pointer to user defined structure. You can append your own data structure using *Node::setInfo()* and *Node::getInfo()*. You are responsible for allocating and de-allocating the user defined data. If you use *ExtCDFG* class, the class is in charge of allocation and de-allocation of user defined data.

Related Functions:

void Node::setInfo(void*);

void* Node::getInfo();

Related Classes:

class ExtCDFG;

Member Functions:

☞***int Node::setId(int id);***

☞***int Node::getId();***

Description:

See member variable *int Node::id*.

☞***void Node::setOp(int op);***

☞***int Node::getOp();***

Description:

It returns operation kind for an operation node (that is, this->getKind() == N_OPER). The operation kind is defined in **token.h**. See member variable *int Node::op* for details.

☞***void Node::setNo(int n); int Node::getNo();***

Description:

See member variable *int Node::_no* for details.

☞***void Node::setName(const char *n);***

Description:

It sets the member variable *String Node::name* to the string contained in the argument '*n*'.

☞***char* Node::getName();***

Description:

It returns char* of the member variable *String Node::name*.

☞***int Node::setType(int t);***

Description:

It set the member variable *short Node::type* to the value in argument 't'. This function is generally not used.

☞***N_TYPE Node::getType();***

***Description*:**

It returns the type of *this Node* which is defined as

```
enum N_TYPE {
        N_NULL=0,          // default
        N_OPER=1,          // operational node
        N_MOD=2,           // module
        N_ITER=4,          // iteration
        N_COND=8           // condition
};
```

See member variables *short Node::type* for details.

☞***int Node::setHwSpeed(int s); int Node::getHwSpeed();***

☞***int Node::setSwSpeed(int s); int Node::getSwSpeed();***

☞***int Node::setHwSwType(int t); int Node::getHwSwType();***

Description:

This function is reserved for user. User can freely use these functions for the purpose of scheduling and binding or for other purposes.

☞***void Node::addPred(Edge *e);***

☞***void Node::addSucc(Edge *e);***

Description:

These routines add new edge to the set of input and output edges (to the member variables *EdgePtrList Node::preds* and *EdgePtrList Node::succs*), respectively.

☞***void Node::subPred(Edge *e);***

☞***void Node::subSucc(Edge *e);***

Description:

These routines subtract an edge 'e' from the set of input and output edges (from the member variables *EdgePtrList Node::preds* and *EdgePtrList Node::succs*), respectively.

☞***void Node::convPred(Edge *old, Edge *cur);***

☞***void Node::convSucc(Edge *old, Edge *cur);***

Description:

These routines replace an edge 'old' to the new edge 'cur' in *EdgePtrList Node::preds* and *EdgePtrList Node::succs*, respectively.

Example:

Following routine shows how to swap two inputs which is connected to 'node'.

```
// Assume 'node' is an operation node with two operands
Subgraph *parent = node->getParent();
Edge *inp0 = node->getPred(0, 1);
Edge *inp1 = node->getPred(1, 1);
// Change the order of edges in parent subgraph
parent->getEdges()->replace(inp0, 1);
parent->getEdges()->replace(inp1, inp0);
parent->getEdges()->replace(1, inp1);
// Change the order of input edges to 'node'
node->convPred(inp0, 1);
node->convPred(inp1, inp0);
node->convPred(1, inp1);
```

☞***EdgePtrList* Node::getPreds();***

☞***EdgePtrList* Node::getSuccs()***

Description:

These routines return the pointer to *EdgePtrList Node::preds* and *EdgePtrList Node::succs*, respectively. Refer to the related member variables for details.

☞***Edge* Node::getPred(int n, int data_flag=0);***

☞***Edge* Node::getSucc(int n, int data_flag=0);***

Description:

These routines return n'th edge connected to the node. If data_flag has value 1, they return n'th input(output) edge whose kind is E_DATA (which corresponds to data transfer edge). Otherwise, they return n'th input(output) edge without checking the kind of an edge. If there exists no n'th edge, they return NULL pointer.

Example:

```
// This example prints two inputs of node 'n'
// Get the first input data edge
Edge *in1 = n->getPred(0, 1);
// Get the second one
Edge *in2 = n->getPred(1, 1);
printf("Two inputs of %s\n", n->getPos());
in1->dump();
in2->dump();
```

☞***int Node::getNumInPort(int edge_kind);***

☞***int Node::getNumOutPort(int edge_kind);***

Description:

These routines return the number of input(output) ports depending on the operation type of *this Node*. If the type of operation(which can be obtained by Node::getOp()) of the node is binary operation, it returns 2, otherwise 1. If the node is not an operational node, they return the number of input(output) edges, where the kind(which can be obtained by Edge::getKind(int)) of the input(output) edge is the same as the value of argument 'edge_kind'.

☞***Subgraph* Node::getParent();***

***Description*:**

It returns the pointer to the parent Subgraph which is stored in 'parent' member variable. Refer to member variable *Subgraph* Node::parent*.

☞***Node* Node::getPNode();***

Description:

It returns the pointer to the nearest parent *Node*. Generally, it returns Node::getParent()->getParent(). When *this Node* has no parent (it is possible for the *CDFG* Class), it returns NULL pointer.

☞***Node* Node::getPModule();***

Description:

It returns pointer to the nearest parent Module which contains *this Node*. If no parent module is found, it returns NULL pointer.

☞***int Node::setMark(int m);***

☞***int Node::getMark();***

Description:

These functions are reserved for the user. You can use these functions during graph traversal – especially for list scheduling or for any other purposes.

☞***char* Node::getPos(char *buf=0);***

Description:

It returns the hierarchy position of *this Node*, where the position is described in the form of string starting from '/', followed by the integer value of node id, and delimited by '/' whenever hierarchy level increases. In the following example, the position of condition node is "/1/1". If the argument variable 'buf' is NULL, the function stores the return value to internal buffer and returns the pointer to the buffer. Otherwise, it stores the return value to 'buf' and returns this pointer.

```
node 0 ...                      // pos: "/0"
node 1 P1 ...                   // pos: "/1"
    subgraph
        node 0 ...              // pos: "/1/0"
        node 1 cond ..          // pos: "/1/1"          -> if ... then ... else
            cond 1
                node 0 ...      // pos: "/1/1/_cond/0
            end
            true 1
                node 0 ...      // pos: "/1/1/_true/0
            end
            false 1
                node 0...       // pos: "/1/1_false/0
            end
        node 2 cond ...         // pos: "/1/2"          -> case ... when ...
            cond 1
```

```
            node 0 ...          // pos: "/1/2/_cond/0
        end
        subgraph 10
            node 0 ...          // pos: "/1/2/_subg0/0
        end
        subgraph 20
            node 0 ...          // pos: "/1/2/_subg1/0
        end
        subgraph 30
            node 0 ...          // pos: "/1/2/_subg2/0
        end
    end
    node 3 loop ...             // pos: "/1/3"                -> while ... loop ...
        iteration 3
            node 0 ...          // pos: "/1/3/_cond/0
        end
        subgraph 3              // pos: "/1/3/0
            node 0 ...
        end
    node 4 MODULE1 ...          // pos: "/1/4"                -> module node
        subgraph 4
            node 0 ...          // pos: "/1/4/0"
        end
end
```

☞***Node* Node::findPos(char *pos);***

Description:

It finds and returns a node whose position is the same as the string contained in 'pos'. If there exist no such node, it returns NULL pointer.

☞***Node* Node::findNode(const char *nname);***

Description:

Find a node whose name (stored in *Node::name*) is the same as 'nname' among the nodes belonging to the *Subgraph* of *this Node*.

☞***Node* Node::getTop();***

Description:

It returns pointer to the top *Node* (*CDFG*).

☞***int Node::getLevel();***

Description:

It returns the hierarchy level of *this Node*. The top *Node* (*CDFG*) has level 0. The hierarchy level increases by one as we step into each hierarchy node (*Condition*, *Iteration*, and *Module*). Following example shows hierarchy level in *CDFG*.

```
edge ...
node 0 ...                  // level = 0
node 1 P1 ...               // level = 0
```

```
subgraph
        node 0 ...                  // level = 1
        node 1 cond                 // level = 1
              condition
                    node  0  ...    // level = 2
              end
               subgraph
                        node  0  ...// level =2
              end
end
```

☞***LinkPtrList* Node::getAttrib();***

Description:

It returns the pointer to *LinkPtrList* which contains the attributes of *this Node*.

Related Classes:

Class LinkPtrList;

Class Link;

☞***Link* Node::findAttrib(char *name);***

Description:

It searches for the attribute whose identifier is the same as 'name'. The attribute is defined as a sequence of string wrapped around by left and right parentheses. We assume the first string of the sequence is the identifier of the attribute.

Example:

This examples find attribute whose identifier is "cstep" and print the value related to the attribute.

```
// This is a part of a textual CDFG file.
node 3 + - - - (cstep 10) (bind mult)

// This is a part of a C++ code.
Link *cstep = node->findAttrib("cstep");
Link *bind = node->findAttrib("bind");
if (cstep)    printf("cstep: %d\n", cstep->item(1)->intValue());
if (bind)     printf("bind: %s\n", bind->item(1)->value());
```

☞***void Node::setTstart(int ts); int Node::getTstart();***

☞***void Node::setTend(int te); int Node::getTend();***

☞***void Node::setPid(int pid); int Node::getPid();***

Description:

These routines are reserved for the user. It can be used during scheduling (*Node::getTstart()*, *Node::setTstart()*, *Node::getTend()*, and *Node::setTend()*) and binding (*Node::setPid()* and

Node::getPid()).

☞***void* Node::getInfo();***

☞***void Node::setInfo(void *);***

Description:

These routines allow the user to attach user defined data structure to a *Node*. See member variable *void* Node::info* for details.

Related Class:

class ExtCDFG;

☞***virtual Subgraph* Node::getCond();***

Description:

It returns pointer to *Subgraph* which contains conditional predicate (*Condition* and *Iteration*). If *this Node* does not contain condition *Subgraph*, it return NULL pointer.

☞***virtual Subgraph* Node::getTrue();***

☞***virtual Subgraph* Node::getFalse();***

Description:

It returns pointer to *Subgraph* which contains true(false) path of *Condition* node. If the type of *this Node* is not N_COND, it returns NULL pointer.

☞***Subgraph* Node::getSubg(int i=0);***

Description:

It returns the i'th subgraph which the node contains. If the node is not an hierarchical node, it returns NULL pointer. It also returns NULL pointer when there does not exist i'th Subgraph. The return value for each combination of the value of 'i' and the type of the node is as follows.

	Condition	Iteration	Module	Node
i = -1	Subgraph* for false path	NULL	NULL	NULL
i =0	Subgraph* for true path	Subgraph*	Subgraph*	NULL
i>0	NULL	Subgraph* or NULL	NULL	NULL

Example:

This example scans all the Subgraphs that *node* contains and print out the contents of each Subgraph.

```
int num_subg = node->getNumSubg();
```

```
for (int i=-1; i < num_subg; i++) {          // Note: index starts from -1
        Subgraph *subgi = node->getSubg(i);
        if (!subgi)          continue;          // No i'th Subgraph exists
        subgi->dump();                          // print out the contents of i'th Subgraph
}
```

☞***virtual int Node::getNumSubg();***

Description:

It returns the number of *Subgraph*s (except for the conditional predicate *Subgraph* which can be obtained by *Subgraph* Node::getCond()*) which belong to *this Node*.

☞***int Node::isHierarchy();***

Description:

It returns 1 if *this Node* is a hierarchy node (*Condition, Iteration*, and *Module*). Otherwise, it returns 0.

☞***virtual void Node::print(int level=1, FILE *fp=stdout, int indent=0);***

Description:

It prints out the contents of *this Node*. The first argument indicates the level of printing out. If the 'level' is zero, it does not print out the contents of its *Subgraph*. Otherwise, it prints out contents of all the nodes and edges in *Subgraph*. The second argument indicates the output file pointer to which the contents of a node will be printed out. The last argument, 'indent', indicates the number of indentation for printing out.

☞***virtual void Node::dump(int level=0);***

Description:

It prints out the contents of *this Node* by using the member function, *void Node::print()*. This function can be used for the purpose of debugging. During debugging your program, you can see the contents of a node by calling this function.

☞***virtual LengthType* Node::findType(const char *typename, int level=0);***

Description:

It searches for a user defined type whose name is the same as the first argument 'typename'. If 'level' is zero, it searches only child *Subgraph*s. Otherwise, it also searches parent *Subgraph*s for the type. If there exists no matching entry in symbol table(*LengthTypePtrList* in *Module*), it returns NULL.

☞***virtual LengthType* Node::findSym(const char *name, int tag=1, int level=0);***

Description:

It searches for a variable whose name is the same as the first argument 'name'. The second argument 'tag' indicates tag matching option during string comparison. It is for the case of indexed variables. If the string stored in 'name' is "ABC(I)" and the 'tag' flag is 1, it compares only the tag name "ABC" excluding the other index name "(I)". The final argument 'level' indicates the range where searching has effect. If the 'level' is 0, it searches only child *Subgraph*s for the variable. However, if the 'level' is 1, it also searches parent *Subgraph*s for the variable. If there exists no matching entry in symbol table (*LengthTypePtrList* in *Module*), it returns NULL.

☞***virtual int Node::getKind();***

***Description*:**

It returns the kind of a node. The kind of the node is dependent on the type of the deriving class which is one of the followings.

```
// For Condition class (getType()== N_COND)
enum COND_KIND {
        COND_IF,                    // if ... then ... else
        COND_CASE                   // case ... when
};
// For Iteration class (getType() == N_ITER)
enum ITER_KIND {
        ITER_NULL=0,
        ITER_FOR,
        ITER_WHILE
};
// For CDFG class (getType() == N_MOD)
enum CDFG_KIND {
        C_NORMAL=1,
        C_HYPER=2,
        C_FLAT_MOD=4,
        C_FLAT_LOOP=8
};
```

☞***virtual void Node::parseInit();***

Description:

This function is called for the initialization of *this Node*, while input CDFG file is parsed. If you derive class from *Node* or you want to append your own data structure using *Node::info* field, you can initialize your own data structure by defining your initialization function here. In addition, you can use *virtual void CDFG::parseInit(Node*)* for this purpose. In this case, you should derive your own CDFG class from base CDFG class and define the member function. The *ExtCDFG* class is devised for such purpose. For details, refer to *ExtCDFG* class.

☞***virtual void Node::copy(Node *src);***

Description:

This function is called for copying contents from another node 'src'. It should be modified if you attach your own data structure to CDFG by derivation or by using *Node::info* field. Refer to *ExtCDFG* class for details.

4.2 *Module* Class

Member Data:

☞***LengthTypePtrList Module::ltlist;***

Description:

It is a kind of symbol table which contains information on each variable such as the type of the variable, bit length, and initial value. Details are described in section 4.7 *LengthType* Class.

☞***Subgraph Module::nsubg;***

Description:

This class corresponds to subgraph belonging to a Module node. If you call Node::getSubg(), we can obtain pointer to 'nsubg'.

Member Functions:

☞***virtual LengthType* Module::findType(const char *type, int level=0);***

☞***virtual LengthType* Module::findSym(const char *name, int tag, int level=0);***

Description:

Refer to *Node* class.

4.3 *Iteration* Class

Member Data:

☞***Subgraph Iteration::ncond;***

Description:

This variable is for a conditional *Subgraph* corresponding to a conditional predicate. If you call Node::getCond() in *Iteration* class, we obtain pointer to this variable.

☞***Subgraph Iteration::nsubg;***

Description:

This variable is for a child *Subgraph* of *Iteration* class which corresponds to a loop body. If you call Node::getSubg(0) in *Iteration* class, we obtain pointer to this variable.

4.4 *Condition* Class

Member Data:

☞***Subgraph Condition::ncond;***

Description:

This variable is for a conditional *Subgraph* corresponding to a conditional predicate. If you call Node::getCond() in *Condition* class, we obtain pointer to this variable.

☞***Subgraph Condition::nfalse;***

Description:

This variable contains *Subgraph* for a false path in conditional branch. If you call *Node::getSubg(-1)* or *Node::getFalse()*, you can obtain pointer to this variable.

☞***SLList<Subgraph*> Condition::ntrues;***

Description:

This variable contains *Subgraph*s for true paths in conditional branch. Generally, there is only one true path (***if*** ... ***then*** ... ***else*** ...). There can be multiple true paths in case of ***case*** construct. You can obtain *i*th *Subgraph* by calling *Node::getSubg(i)* in *Condition* class. Each true path corresponds to each ***when*** construct in VHDL. In this case the name of each *Subgraph* corresponds to the condition in which the ***when*** construct is activated.

4.5 *Edge* Class

Member Data:

☞***Node * Edge::pred;***

Description:

This variable indicates a pointer to predecessor *Node*.

Related Functions:

Node Edge:getPred();* *Node* Edge::setPred(Node *p);*

☞***Node * Edge::succ;***

Description:

This variable indicates a pointer to successor *Node*. There is only one successor if the kind of the top CDFG is C_NORMAL. If the kind of the top CDFG is C_HYPER, there can be multiple successors which are stored in *NodePtrList Edge::_succs* (experimental).

Related Functions:

Node Edge::getSucc();* *Node* Edge::setSucc(Node *s);*

☞***String Edge::name;***

Description:

This variable indicates the name of an edge. The name corresponds to the variable name explicitly given in VHDL, the reserved, name or the temporary name implicitly given by ***cdfggen***. The naming convention in the edge's name and type is as follows.

name	**type**	**description**
*_tmp_**	-	Temporary variable. The variable is stored in symbol table as LengthType in parent node.
depend	*bool*	Reserved variable name and type for representing control dependencies
others	others	Variable name and type explicitly given in VHDL

Related Functions:

*void Edge::setName(const char *n);*

char Edge::getName();*

☞***String Edge::type;***

Description:

This variable indicates the type name of an edge. It corresponds to the type of the data, reserved type by ***cdfggen***. For the naming convention on edge's name and type, refer to *String Edge::name*.

Related Functions:

char Edge::getType();*

*void Edge::setType(const char *t);*

☞***int Edge::weight;***

Description:

This variable indicates the weight of the data transfer by this *Edge*. If the transferred data is array type, the value of *weight* can be larger than 1, otherwise, the value is always 1.

Related Functions:

void Edge::setWeight(int w);

int Edge::getWeight();

☞***int Edge::bits;***

Description:

This variable indicates the number bits transferred through this *Edge*. It is computed as the product of *weight* and the number of bit for *Edge::type*. The number of bits corresponding to *Edge::type* is defined in symbol table(*Module::LengthTypePtrList*) of parent node.

Related Functions:

void Edge::setBits(int b);

int Edge::getBits();

☞***int Edge::mark;***

Description:

This variable is reserved for the user. It can be used during graph traversal for marking visited edges.

Related Functions:

void Edge::setMark(int m);

int Edge::getMark();

☞***short Edge::kind;***

Description:

This variable represents the kind of this *Edge*. The kind of each edge is defined as enumeration type, EDGE_KIND, which is described as follows.

```
enum EDGE_KIND {
        E_NULL = 0,
        E_DATA = 1,
        E_CTRL = 2,
        E_ENDLOOP = 4,
        E_DEPEND = 8,
        E_EVENT = 0x10,
        E_RETURN = 0x20
};
```

Related Functions:

int Edge::getKind(int calc=0);

☞***int Edge::port;***

Description:

This variable is reserved for the user.

Related Functions:

void Edge::setPort(int p);

int Edge::getPort();

☞***LinkPtrList Edge::_attrib;***

Description:

This variable contains attributes attached to this *Edge*. For the definition and use of attribute field, refer to section 2.2, *LinkPtrList Node::_attrib*, and *Link* Node::findAttrib(const char*).*

Related Functions:

LinkPtrList Edge::getAttributes();*

Link Edge::findAttrib(const char *n);*

☞***int Edge::_no;***

Description:

This variable represent the line number in textual CDFG file.

Related Functions:

void Edge::setNo(int n);

int Edge::getNo();

☞***int Edge::tstart, tend;***

Description:

This variables are reserved for the user. They can be used during scheduling and binding.

Related Functions:

int Edge::getTstart(); *int Edge::setTstart(int ts);*

int Edge::getTend(); *int Edge::setTend(int te);*

☞***void* Edge::info;***

Description:

This variable contains the pointer to user defined data structure. For details, refer to *void* Node::info*, *void* Node::getInfo()*, and *ExtCDFG* class.

Related Functions:

*void Edge::setInfo(void *i);*

void Edge::getInfo();*

Member Functions:

☞***Node* Edge::setPred(Node* p); Node Edge::getPred();***

Description:

These routines set and return the pointer to a predecessor node.

☞***Node* Edge::setSucc(Node* s); Node Edge::getSucc();***

Description:

These routines set and return the pointer to a successor node.

☞***void Edge::setPort(int p); int Edge::getPort();***

Description:

These routines set and return the value of *Edge::port*. They are reserved for the user.

☞***int Edge::setMark(int m); int Edge::getMark();***

Description:

These routines are reserved for the user. They can be used during scheduling and binding.

☞***void Edge::setName(const char *n); char * Edge::getName();***

Description:

These routines set and get the name of an edge. For details, refer to *String Edge::name*.

☞***void Edge::setType(const char* t); char* Edge::getType();***

Description:

These routines set and get the type of an edge. For details, refer to *String Edge::type*.

☞***int Edge::setWeight(int w); int Edge::getWeight();***

Description:

These routines set and get the weight of an edge. For details, refer to *int Edge::weight.*

☞***int Edge::setBits(int b); int Edge::getBits();***

Description:

These routines set and get the number of bits of an edge. For details, refer to *int Edge::bits*.

☞***int Edge::getKind(int calc=0);***

Descripton:

This routine returns the kind of an edge. If the argument 'calc' has non-zero value, it again check the name and type of the edge to return proper edge kind presented in *int Edge::kind*. Since it takes time for the checking, we usually use the result obtained in previous call. We use such scheme by setting the argument 'calc' to 0.

☞***int Edge::isConst()*;**

Description:

This routine tells if the data transferred by *this Edge* is constant type. It checks a few leading characters of *String Edge::Name* for checking. Thus, it may return incorrect result. However, it always returns correct result for the cdfg file generated by *cdfggen*, because syntax on the variable naming convention is checked during parsing input VHDL description.

☞***void Edge::setNo(int n)*;** ***int Edge::getNo();***

Description:

These routines set and get the line number in the input CDFG file. For details, refer to *int Edge::_no*.

☞***LinkPtrList* Edge::getAttrib();***

Description:

It returns pointer to *LinkPtrList* which contains the list of *Link*s which correspond to attribute field in the textual CDFG format. For details, refer to section 2.2, *Link* Node::findAttrib(char *n)*, and *LinkPtrList Node::_attrib*.

☞***Link* Edge::findAttrib(char *name)***

***Description*:**

It searches for the attribute whose identifier is the same as 'name'. The attribute is defined as a sequence of string wrapped around by left and right parentheses. We assume the first string of the sequence is the identifier of the attribute.

***Example*:**

The following example shows how to access the user defined attribute field whose identifier is "bind".

```
// This is a part of input CDFG file
edge 0 1 ...  (bind reg1)
edge 1 2 ...  (bind reg2)

// This is a part of C++ code
Subgraph *subg = mycdfg.getSubg();
EdgePtrList &edges = *subg->getEdges();
for (Pix pi=edges.first(); pi; edges.next(pi)){
        Edge *e = edges(pi);
        Link *bind_attr = e->findAttrib("bind");
        if (bind_attr)          printf("%s\n", bind_attr->item(1)->value());
}
```

☞***void Edge::setTstart(int ts);*** ***int Edge::getTstart();***

***Description*:**

These routines are reserved for the user to set and get *int Edge::tstart*.

☞***void Edge::setTend(int ts);*** ***int Edge::getTend();***

***Description*:**

These routines are reserved for the user to set and get *int Edge::tend*.

☞***void Edge::setInfo(void *i)*;** ***void* Edge::getInfo()*;**

***Description*:**

These routine set and get the pointer to user defined data structure stored in *void* Edge::info*.

☞***int Edge::compare*(*Node *pred, Node *succ, const char *name, int wildcmp=0, const char *type=0, int kind=0*);**

***Description*:**

This function returns one if the contents of *this Edge* is the same as all the arguments. The matching pair (Argument and content in *Edge*) is summarized as follows

Argument	**Argument Value**	***Edge***	**Comparison**
prev	NULL	-	No
prev	non-zero	Edge::prev	Yes
succ	NULL	-	No
succ	non-zero	Edge::succ	Yes
name	NULL	-	No
name	non-zero	Edge::name	Yes
wildcmp	0	Edge::name	Full comparison
wildcmp	1	Edge::name	Partial comparison
type	NULL	-	No
type	non-zero	Edge::type	Yes
kind	0	-	No
kind	non-zero	Edge::kind	Yes

The partial comparison means comparison of only *N* leading characters where *N = strlen(name).*

☞***virtual void Edge::print(FILE *fp=stdout, int i=0);***

Description:

This routine prints out the contents in *this Edge*. The first argument indicates the file pointer to which output will be printed. The second argument indicates the size of indentation.

☞***virtual void Edge::parseInit();***

Description:

This function is called for the initialization of *this Edge*, while input CDFG file is parsed. If you derive class from *Edge* or you want to append your own data structure using *Edge::info* field, you can initialize your own data structure by defining your initialization function here. In addition, you can use *virtual void CDFG::parseInit(Edge*)* for this purpose. In this case, you should derive your own CDFG class from base CDFG class and define the member function. The *ExtCDFG* class is devised for such purpose. For details, refer to *ExtCDFG* class.

☞***virtual void Edge::copy(Edge *src);***

Description:

This function is called for copying contents from another node 'src'. It should be modified if you attach your own data structure to CDFG by derivation or by using *Edge::info* field. Refer to *ExtCDFG* class for details.

4.6 *Subgraph* Class

Member Data:

☞***int Subgraph::id;***

Description:

This variable contains the id of each *Subgraph*. It is generally not used.

Related Functions:

int Subgraph::setId(int i);

int Subgraph::getId();

☞***String Subgraph::pname;***

Description:

This variable contains the primary name of *this Subgraph*. It usually has the string "subgraph" in most cases. The following table shows the pname in accordance with the type of parent subgraph.

Parent	**Subgraph Type**	***Subgraph::pname***
Condition	conditional	"cond"
Condition	true path (**if** ... **then** ... **else**)	"true"
Condition	false path	"false"
Condition	true path (**case** ... **when** ...)	"subgraph"

Iteration	conditional	"iteration"
Iteration	loop body	"subgraph"
Module	child *Subgraph*	"subgraph"

The following example shows how the *Subgraph::pname* is determined. In the examples, the string in bold face indicates the value of *Subgraph::pname*.

```
node 0 ...
node 1 P1 ...
    subgraph 1
        node 0 ...
        node 1 cond ..          -> if ... then ... else
            cond 1
                node 0 ...
            end
            true 1
                node 0 ...
            end
            false 1
                node 0...
            end
        node 2 cond ...         -> case ... when ...
            cond 1
                node 0 ...
            end
            subgraph 10
                node 0 ...
            end
            subgraph 20
                node 0 ...
            end
            subgraph 30
                node 0 ...
            end
          end
        node 3 loop ...         -> while ... loop ...
            iteration 3
                node 0 ...
            end
            subgraph 3
                node 0 ...
            end
        node 4 MODULE1 ...      -> module node
            subgraph 4
                node 0 ...
            end
    end
```

Related Functions:

*void Subgraph::setPName(const char *n);*

char Subgraph::getPName();*

☞***String Subgraph::name;***

Description:

This variable contains the secondary name of *this Subgraph*. It has meaning only when the

type of parent *Node* is N_COND and the kind of the node is COND_CASE. This variable contains the case when each 'when' statement is activated. Following example shows how the value of *Subgraph::name* is determined.

```
// This is a part of an input VHDL file
case input =>
        when "1111" => ...
        when "1110" => ...
        ...
end case;

// This is a part of a textual CDFG file
node 1 cond ...
    cond 1
     end
    subgraph  "1111"          // Subgraph::name = "1111"
     end
    subgraph  "1110"          // Subgraph::name = "1110"
     end
```

Related Functions:

*void Subgraph::setName(const char *n);*

char Subgraph::getName();*

☞***NodePtrList Subgraph::nlist;***

Description:

This variable is the list of nodes that belong to *this Subgraph*. The nodes belonging to this *Subgraph* are those defined between *subgraph* (or other equivalent primary name defined in *Subgraph::pname*) and *end* keywords as shown in the following example.

```
node 1 TOP - - -
    subgraph 1
        node 0 source - - -    // element in Subgraph::nlist
        node 1 + - - -         // element in Subgraph::elist
     end
```

The user does not need to de-allocate the *Node* classes which are added to *this Subgraph* by the user (by way of *Subgraph::addNode()* or other methods). The destructor of the *Subgraph* class automatically de-allocates all the elements in *Subgraph::nlist* by using ***delete***.

Related Functions:

*void Subgraph::addNode(Node *n);* *void Subgraph::subNode(Node *n);*

NodePtrList Subgraph::getNodes();*

☞***EdgePtrList Subgraph::elist;***

Description:

This variable is the list of edges that belong to *this Subgraph*. The user does not need to de-allocate the *Edge* classes which are added to *this Subgraph* by the user (by way of *Subgraph::addEdge()* or other methods). The destructor of the *Subgraph* class automatically de-allocates all the elements in *Subgraph::elist* by using ***delete***.

Related Functions:

*void Subgraph::addEdge(Edge *e);* *void Subgraph::subEdge(Edge *e);*
EdgePtrList Subgraph::getEdges();*

☞***AssignPtrList Subgraph::alist;***

Description:

This variable is the list of pointers to *Assign* classes. The *Assign* class maps I/O signal names between boundary of component instantiation and function call. For more details, refer to *Assign* class. The user does not need to de-allocate the *Assign* classes which are added to *this Subgraph* by the user (by way of *Subgraph::addAssign()* or other methods). The destructor of the *Subgraph* class automatically de-allocates all the elements in *Subgraph::alist* by using ***delete***.

Related Functions:

*void Subgraph::addAssign(Assign *a);* *void Subgraph::subAssign(Assign *a);*
AssignPtrList Subgraph::getAssigns();*

☞***Node * Subgraph::parent;***

Description:

This variable indicates the pointer to parent *Node*. The parent node can be one of *Module, Condition, Iteration,* and *CDFG* classes.

Related Functions:

Node Subgraph::setParent(Node n);* *Node* Subgraph::getParent();*
Node Subgraph::getTop();*

☞***Node ** Subgraph::map;***

Description:

This variable plays the role of mapping table between a node's id and the pointer to the node. It is used in *Node* Subgraph::node(int i).* Since child nodes are stored in *NodePtrList Subgraph::nlist*, it takes time for find a node in the list. This table is implemented for the

purpose of fast node searching by way of the node's id which can be used as an index of the mapping table.

Related Functions:

Node Subgraph::node(int id);*

void Subgraph::mapNode(int parse=1);

☞***int Subgraph::num_node;***

Description:

This variable indicates the number of node within *this Subgraph*.

☞***void * Subgraph::info;***

Description:

This variable indicates the pointer to user defined structure. You can append your own data structure using *Subgraph::setInfo()* and *Subgraph::getInfo()*. You are responsible for allocating and de-allocating the user defined data. If you use *ExtCDFG* class, the class is in charge of allocation and de-allocation of user defined data.

Related Functions:

void Subgraph::setInfo(void);*

void Subgraph::getInfo();*

Member Functions:

☞***int Subgraph::setId(int i);*** ***int Subgraph::getId();***

Description:

These routines set and get the id (stored in *Subgraph::id*) of *this Subgraph.* The id of the subgraph is usually not used.

☞***void Subgraph::setPName(const char* n); char* Subgraph::getPName();***

Description:

These routines set and get the primary name of *this Subgraph* (stored in *Subgraph::pname*). For details, refer to *String Subgraph::pname*.

☞***void Subgraph::setName(const char*n);*** ***char* Subgraph::getName();***

Description:

These routines set and get the secondary name (stored in *Subgraph::name*) of *this Subgraph.*

☞***void Subgraph::addNode(Node *n);***

Description:

This routine appends a *Node* 'n' to *this Subgraph*, where the node 'n' is stored in *Subgraph::nlist*.

☞***void Subgraph::addEdge(Edge *e);***

Description:

This routine appends an *Edge* 'e' to *this Subgraph*, where the edge 'e' is stored in *Subgraph::elist*.

☞***void Subgraph::addAssign(Assign *a);***

Description:

This routine appends an *Assign* 'a' to *this Subgraph*, where the assign 'a' is stored in *Subgraph::alist*.

☞***void Subgraph::subNode(Node *n);***

Description:

This routines subtract a node 'n' from *this Subgraph* (from the member data *Subgraph::nlist*).

☞***void Subgraph::subEdge(Edge *e);***

Description:

This routines subtract an edge 'e' from *this Subgraph* (from the member data *Subgraph::elist*).

☞***void Subgraph::subAssign(Assign *a);***

Description:

This routines subtract an assign 'a' from *this Subgraph* (from the member data *Subgraph::alist*).

☞***NodePtrList* Subgraph::getNodes();***

Description:

It returns the pointer to the member data *Subgraph::nlist*.

☞***EdgePtrList* Subgraph::getEdges();***

Description:

It returns the pointer to the member data *Subgraph::elist*.

☞***AssignPtrList* Subgraph::getAssigns();***

Description:

It returns the pointer to the member data *Subgraph::alist*.

☞***Node* Subgraph::setParent(Node* n); Node* Subgraph::getParent();***

Description:

These routines set and get the parent *Node* of *this Subgraph.*

☞***Node* Subgraph::getTop();***

Description:

It returns the pointer to the top *Node*(that is, *CDFG*).

☞***Node* Subgraph::node(int id);***

Description:

It finds a node whose id is the same as the argument 'id' and returns the pointer to the node. It uses the mapping table *Subgraph::map* for fast searching. In the CDFG, there are two reserved ID's, -1 for sink and 0 for source node. Therefore, the valid range of the first argument 'id' is between –1 and the value returned by *Subgraph::getMaxNodeId()*.

☞***int Subgraph::empty();***

Description:

It returns 0 if this *Subgraph* is empty. Otherwise, it returns 1.

☞***int Subgraph::getNumNodes(int level=0);***

☞***int Subgraph::getNumEdges(int level=0);***

Description:

It returns the number of nodes(edges) within *this Subgraph*. It returns different values according to the value of argument 'level'. Following table summarizes the relation between the value of 'level' and the return value.

level	Return Value
0	Number of nodes(edges) within *this Subgraph* excluding all the child nodes in different hierarchy.

1	Number of nodes(edges) within *this Subgraph* including all the child nodes in different hierarchy.
2	The return value is similar to that returned when 'level' is 1 except that duplicated I/O edges and source/sink nodes are not counted.

☞***int Subgraph::getMaxNodeId();***

Description:

This function returns the maximum value of node's ID belonging to *this Subgraph*. You can use this function to assign ID to a node which is added in your program. Following example shows how to insert your own node to *this Subgraph*.

```
// 'n' is your own node which will be added to the graph
n = new Node(subg->getMaxNodeId()+1); //Set the node's ID such that there is no duplication.
...     / Build your own Node here.
subg->addNode(n);
```

☞***void Subgraph::parse(FILE *fp);***

Description:

This routine parses the file indicated by the file pointer 'fp'. Nodes in upper level use it. The end user need not use this function.

☞***void Subgraph::print(int level=1, FILE *fp=stdout, int i=0);***

Description:

It prints out the contents of *this Subgraph*. The effect of the three arguments is the same as in *void Node::print(int, FILE*, int)*.

☞***void Subgraph::dump(int level=0);***

Description:

This routine calls the member function *Node::print(level, stdout, 0)*. It can be used in the course of source code debugging.

☞***void Subgraph::mapNode(int parse=1);***

Description:

This function builds mapping table implemented as *Node** Subgraph::map*. If the argument is '1', this function also connects each edge to each node based on the intermediate information

in the course of parsing. It the argument is '0', it builds Node** Subgraph::map and re-connects an edge to a node based on the predecessor and successor information contained in each edge. The second option (parse = 0) is useful when we want to modify connection of the graph. To modify the graph, we first change the pointer to a predecessor and a successor of each node, then we still have to change the input and output edge connected to the predecessor and the successor. This job is tedious and error-prone, because we have to preserve the order of input edges connected to each node while modifying the graph. Simpler method is to first change the pointer to a predecessor and a successor of each node and call the *Subgraph::mapNode()* with the *parse* argument set to 0.

☞***void Subgraph::getType();***

Description:

This function returns the type of a subgraph, where subgraph type is defined as

```
enum SUBG_T {
        ST_NULL = 0,
        ST_COND,        // Subgraph for conditional predicate (in if ... then ... else and while)
        ST_ITER,        // Subgraph for loop body
        ST_TRUE,        // Subgraph for true path in if ... then ... else
        ST_FALSE,       // Subgraph for false path in if ... then ... else
        ST_SUBG         // Normal Subgraph
};
```

☞***void Subgraph::recurCall(void (*func)(Node , void *), void *arg);***

Description:

This function calls a function transferred by the first argument *func* for each node in a sub-graph such that the first argument to *func* is pointer to each node in the sub-graph and the second one is pointer to the user argument transferred to *Subgraph::recurCall* as second argument *arg*. The function *func* is called while recursively scanning the hierarchical sub-graphs. Following example show how to set the marking value of each node belonging to the sub-graph *subg*.

```
void SetMark(Node *n, void *arg) {
        int markvalue = * (int*)arg;
        n->setMark(markvalue);
}

void SetSubgMark(Subgraph *subg, int markvalue) {
        subg->recurCall(SetMark, &markvalue);
}
```

☞***void Subgraph::getInfo(); void Subgraph::setInfo(void *);***

Description:

This routines get and set the *Subgraph::info* variable. For details refer to *ExtCDFG* class.

4.7 LengthType Class

Member Data:

☞***String LengthType::name;***

Description:

This variable represents the name of symbol where each symbol name corresponds to the name of each data transfer edge in the CDFG.

☞***String LengthType::type;***

Description:

This variable represents symbolic name of each data symbol. It may be reserved type such as "INTEGER" and "REAL", or user defined data type.

☞***int LengthType::bits***

Description:

This variable indicates the number of bits for representing the data.

☞***short LengthType::type_kind;***

Description:

This variable indicates the kind of data type which is defined as

```
enum LT_KIND {
        LT_NULL = 0,
        LT_IN = 1,              // input port
        LT_OUT=2,               // output port
        LT_INOUT=4,             // inout port
        LT_SUBTYPE=8,           // subtype
        LT_ARRAY=0x10,          // Array Type
        LT_CONSTANT=0x20,       // Constant type
        LT_INDEX                // Index type
};
```

There are two categories of *LengthTypes* based on the value of *LengthType::type_kind*. If the value is one of *LT_IN, LT_OUT, LT_INOUT,* and *LT_CONSTANT*, the symbol in the *LengthType* class corresponds to variable, signal, and constant in VHDL. Otherwise, it is a supplementary

symbol table entry which is related to user defined type(*LT_SUBTYPE*) or array type(*LT_ARRAY*).

☞***int LengthType::left, right;***

Description:

This variables represent the range (left range and right range) of a data.

☞***String LengthType::constants;***

Description:

This variable contains the (initial) value of the constant type symbol.

☞***Module* LengthType::pmodule;***

Description:

This variable indicates the pointer to the parent *Module* to which this symbol table (*LengthType*) belongs.

☞***LengthType* LengthType::child;***

Description:

This variable holds pointer to the child *LenghType*. The child *LengthType* exists only when the kind(in *LengthType::type_kind*) of a parent *LengthType* is *LT_SUBTYPE*. The child *LengthType* corresponds to the base type in the definition of subtype in VHDL.

☞***int LengthType::port;***

☞***int LengthType::mark;***

Description:

This variables are reserved for the user.

☞***LinkPtrList LengthType::_attrib;***

Description:

This variable corresponds to attribute field in the input CDFG file. For details refer to *Node::findAttrib()*;

Member Functions:

☞***void LengthType::setPort(int p);*** ***int LengthType::getPort();***

☞***void LengthType::setMark(int m);*** ***int LengthType::getMark();***

Description:

These functions are reserved for the user. They give methods to access the member variables *LengthType::port* and *LengthType::mark*.

☞***Module* getPModule();***

Description:

This function returns the value stored in *LengthType::pmodule*.

☞***int LengthType::isSym();***

Description:

This function returns 1 if the symbol in this *LengthType* corresponds to real symbolic data. That is, the symbol name represents the name of real data such as variable, signal and constant in VHDL.

☞***LinkPtrList* LengthType::getAttrib();***

Description:

It returns the pointer to *LengthType::_attrib* variable.

☞***Link* LengthType::findAttrib(char *name);***

Description:

It finds the attribute fields attached to the *LengthType*. For details refer to *Node::findAttrib()*.

☞***void LengthType::print(FILE *fp, int indent);*** ***void LengthType::dump();***

Description:

It prints out the contents in *this* class. It can be used for debugging.

4.8 Assign Class

Member Data:

☞***String Assign::actual;***

Description:

This variable contains actual name of interface variable. The interface variable indicates the argument of function and procedure or port of an entity. The actual name means the symbolic name of a variable used in upper hierarchy. On the other hand, the symbolic name of a variable used in current hierarchy(or lower hierarchy) is referred as formal name.

☞***String Assign::formal;***

Description:

This variable contains formal name of interface variable. The interface variable indicates the argument of function and procedure or port of an entity. The formal name means the symbolic name of a variable used in current hierarchy. On the other hand, the symbolic name of a variable used in upper hierarchy is referred as actual name.

Member Functions:

☞***void Assign::print(FILE *fp, int indent); void Assign::dump();***

Description:

These functions print out the contents of the class. It can be used for debugging.

4.9 PtrList<T>

The *PtrList<T>* class is a linked list of pointers. It is used to manage lists of *Node* classes and *Edge* classes as well as *Assign*, *LengthType*, and *Link* classes. The class is derived from *SLList<T>* class which is included in GNU g++ library. The *SLList<T>* class provides singly linked data structure where we don't need to explicitly build data structure for linked list. In *PtrList<T>* class, we added several member functions for the ease of manipulating list of pointers: for example, searching pointer, replacing existing pointer, finding i'th element, removing matching pointers, etc.

Member Functions:

☞***T* PtrList<T>::isIn(T* d, Pix *last_pos=0);***

Description:

This function searches the pointer '*d'* in the list and returns the found pointer ('*d*') if it is in the list, and NULL pointer, otherwise. The second argument 'last_pos', which is usually NULL pointer, indicates the position in the list from which searching should start and to which the searching is performed at the last call of the function. Following example shows how to find all the wanted elements in the list.

```
PtrList<Node> list;
Node *a = ... ;
Pix last_pos = 0;
while (list.isIn(a, &last_pos)) {
        // perform some job here...
}
```

☞***T* PtrList<T>::sub(T* d);***

Description:

This function subtracts a pointer in the list whose value is the same as that of 'd'. It returns the subtracted pointer in case of success, otherwise, it returns NULL pointer. It subtracts one element for each call. Therefore, to remove all the pointers same as 'd' in the list, you should repeat calling the function until it returns NULL. Following example shows how to remove all the pointers whose value is same as the of 'a'.

```
PtrList<Node> list;
Node *a = ...;
while (list.sub(a));
```

☞***void PtrList<T>::replace(T *old, T *cur);***

Description:

This function replaces existing pointers which are the same as *'old'* by '*cur'*. It replaces all the matching elements in the list.

☞***T* PtrList<T>::item(int num);***

Description:

This function returns *num'th* element in the list. The indexing number starts from zero. If the index is out of the length of the list, it returns NULL.

4.10 NodePtrList

Member Functions:

☞***Node* NodePtrList::node(int id);***

Description:

This function searches a *Node* by its ID and returns the result. If no matching *Node* is found, it returns NULL, otherwise, it returns the pointer to the found *Node*.

☞***Node* NodePtrList::isIn(const char *name);***

Description:

This function searches for a *Node* by its name and returns the result. If no matching *Node* is found, it returns NULL, otherwise, it returns the pointer to the found *Node*.

☞***void NodePtrList::dump(int level=0);***

Description:

This function prints out the contents of the elements in the list. The argument *'level'* has same meaning in *Node::print()* function.

☞***void NodePtrList::recurCall(void (*func)(Node*, void*), void *flag=0);***

Description:

This function calls a function *func* transferred as the first argument such that the first argument of *func* is each element in the list and the second argument of *func* is the *flag* transferred as the second argument of the function. For details, refer to *Subgraph::recurCall()*.

4.11 EdgePtrList

Member Functions:

☞***Edge* EdgePtrList::isIn(Node *pred, Node *succ, const char *name, int wildcmp=0, const char *type=0, int kind=0);***

Description:

This function searches for an *Edge* by *Edge::pred*, *Edge::succ*, *Edge::name, Edge::type,* and *Edge::getKind()*. It returns pointer to the first found *Edge* in the list. This table summarizes the relation between the arguments of the function and the contents of an *Edge* class. If the pointer value of an argument is NULL, it does not compare the corresponding variable in *Edge* to find a matching *Edge*. The fourth argument, *wildcmp*, is an option for comparing *Edge::name*. If this value is 1, this function compares first *N* characters in *name* where *N=strlen(name)*. For details, refer to *Edge::compare()*.

arguments	*pred*	*succ*	*name*	*wildcmp*	*type*	*kind*
matching variable in *Edge*	*Edge::pred*	*Edge::succ*	*Edge::name*		*Edge::type*	*Edge::getKind()*

☞***int EdgePtrList::getNumData();***

Description:

This function computes and returns the number of *Edges* whose kind(returned by *Edge::getKind()*) is E_DATA.

☞***void EdgePtrList::find(EdgePtrList &dest, Node *pred, Node *succ, const char *name, int wildcmp=0, const char *type=0, int kind);***

Description:

This function performs almost same operation as *Edge::isIn()*. Even though *Edge::isIn()* searches for only one matching *Edge*, this function searches for all the matching *Edges* and stores them to the *'dest'*. For details on matching scheme, refer to *EdgePtrList::isIn()*.

☞***void EdgePtrList::dump();***

Description:

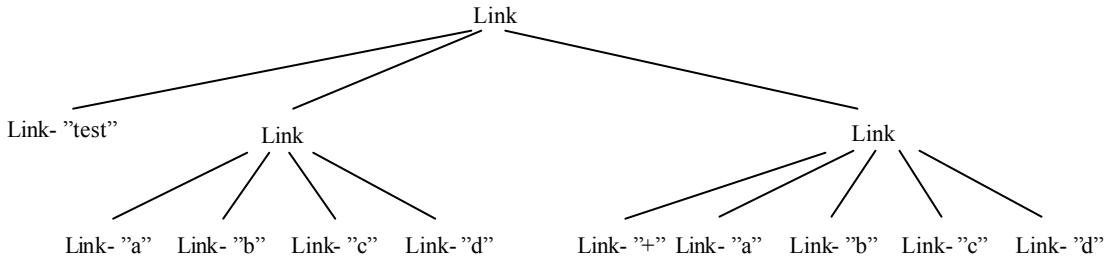
This function prints out the contents of all the *Edges* in the list. It can be used for debugging.

4.12 Link

The *Link* and *LinkPtrList* classes are kind of parser classes which build internal data structure from a set of string described in LISP form. The LIST format string is represented as a set of string wrapped around by left and right parentheses. Following example shows an examples.

(test (a b c d) (+ a b c d))

The *Link* class corresponds to a **leaf string** (such as "test"), a **leaf list** (such as (a b c d)), or a **hierarchical list** (such as (test (a b c d) (+ a b c d))). When a *Link* class parses the example string, it builds internal data structure as shown in the following figure.



In CDFG, attribute field is described in LISP format. Thus they are stored in LinkPtrList as *Node::_attrib* , *Edge::_attrib,* and *LengthType::_attrib*.

Member Data:

☞***String Link::_val*;**

Description:

This variable contains the **leaf string**. It has no value if the *Link* class is not a leaf string but a **leaf list** or a **hierarchical list** If a *Link* is a list elements of the list is stored in the base class *LinkPtrList*.

☞***int Link::_no;***

Description:

This variable indicates the line number at which the string appears in the input file. It can be used for printing error message.

Member Functions:

☞***int Link::isList();***

Description:

This function returns 1 if the class is a list *Link* (a **leaf list** or a **hierarchical list**) *Link*. Otherwise, it returns 0.

☞***int Link::compare(const char *str, int caseflag=0);***

Description:

This function is applicable only when the class corresponds to a **leaf string**. This function returns 1 if the string value stored in *Link::_val* is the same as the first argument *'str'* . Otherwise, it returns 0. The second argument *'caseflag'* is an option for string comparison. If the value of this argument is 1, it performs case-sensitive comparison.

☞***int Link::compare(Link *l, int caseflag=0);***

Description:

This function is applicable only when the class corresponds to a **leaf string**. This function returns 1 if the string value stored in *Link::_val* is the same as that of the link *'l'* which is the first argument of the function. Otherwise, it returns 0. The second argument *'caseflag'* is an option for string comparison. If the value of this argument is 1, it performs case-sensitive comparison.

☞***int Link::compareHeader(const char *hdr, int caseflag=0);***

Description:

This function is applicable only when the class corresponds to a **leaf list** or a **hierarchical list** where the first element in the list is a **leaf string**. We refer the first element of a list as a **header** of the list. This function compares the value of the header with that of the first argument *'hdr'*. If they have the same value, it returns 1. Otherwise, it returns 0. The second argument *'caseflag'* is an option for string comparison. If the value of this argument is 1, it performs case-sensitive comparison.

☞***int Link::setNo(int n);*** ***int Link::getNo();***

Description:

These functions set and get the line number stored in *Link::_no*.

☞***char* Link::value();***

Description:

This function is applicable only when the class contains a **leaf string**. It returns raw string value stored in *Link::_val*.

☞***int Link::intValue();***

Description:

This function is applicable only when the class contains a **leaf string**. It returns integer value of the string stored in *Link::_val.*

☞***float Link::floatValue();***

Description:

This function is applicable only when the class contains a **leaf string**. It returns float value of the string stored in *Link::_val.*

☞***void Link::setValue(const char *fmt, ...);***

Description:

This function is applicable only when the class corresponds to a **leaf string**. It set the string value of *Link::_val* to the string value in *'fmt'*. The first argument *'fmt'* can be described in the same way as *printf* or *scanf*.

☞***int Link::length();***

Description:

This function returns the number of elements in the class. If the class is a **list** class it returns the number of elements in the list. Otherwise (in case of **leaf string**), it returns 1. It counts the number of elements in the same level. That is, it regards the elements of **list** type *Link* as one element. It does not step into the **list** type *Link* class and does not count the number of elements in the class.

☞***int Link::parse(FILE *fp, int lno=0);***

Description:

This function builds a *Link* structure from an input text file. The first argument *fp* indicates a file pointer to the input text file and the second one indicates the initial value of line number which will be stored in *Link::_no*.

☞***int Link::parse(char *buf);***

Description:

This function builds a *Link* structure from a string. The first argument *buf* indicates an input string described in LISP format.

☞***int Link::print(FILE *fp, int indent=0, int fmt=0);***

Description:

This function prints out contents in the class to the file pointer *fp*. It can be used for debugging.

☞***void Link::dump();***

Description:

This function prints out contents in the class to *stdout*. It can be used for debugging.

4.13 LinkPtrList

The *LinkPtrList* class is a base class of *Link* where it also contains the list of pointers to *Link* classes.

Member Functions:

☞***int LinkPtrList::parse(char *token, char *buf=0, FILE *fp=0, int lno=0);***

Description:

This function builds a *Link* class from an input string *buf* and *token* or from an input file *fp*. The built *Link* class is stored as an element of *this* class. This function is internally called in *LinkPtrList::parseList(FILE*, int)* , *LinkPtrList::add(const char *,...)* and *Link::parse(char*)*.

☞***int LinkPtrList::parseList(FILE *fp, int lno=0);***

Description:

This function builds a *Link* class from an input text file *fp*. The second argument *lno* is the initial value of the line number stored in *Link::_no*.

☞***Link* LinkPtrList::findList(const char *hdr, int caseflag=0, Pix *start=0);***

☞***Link* LinkPtrList::findList(Link *l, int caseflag=0, Pix *start=0);***

Description:

These functions find a **list** type *Link* which matches the name of header stored in *hdr* and *l->_val*. The second argument *caseflag* is a string comparison option. If the value of the argument is 1 it performs case-sensitive string comparison. The last argument *start* is the position in the list from which searching starts or to which the position in the list at the last call is stored. You can find all the matching *Link* classes in the list by calling the function repeatedly as shown in the following example.

```
LinkPtrList list;
...
Pix last_pos = 0;
Link *cur;
while ((cur = list.findList("cstep", 0, &last_pos))) {
        ...
}
```

☞***Link* LinkPtrList::findValue(const char *value, int caseflag=0, Pix *start=0);***

Description:

This function finds a **leaf string** type *Link* class whose value (*Link::_val*) is the same as *value*. The second argument *caseflag* is an option for string comparison. If the value of the argument is 1, it performs case-sensitive string comparison.

☞***int LinkPtrList::indexList(const char *hdr, int caseflag=0, int start=0);***

☞***int LinkPtrList::indexList(Link *l, int caseflag=0, int start=0);***

Description:

These functions search for a **list** type *Link* class whose **header** is the same as *hdr* or *l->_val*. Then, it returns the index (the position in the list where the first element's index is 0) of the *Link* class in the list. The second argument *caseflag* is a string comparison option. If the value of the argument is 1, it performs case-sensitive string comparison. The last argument is the position in the list from which searching starts or to which the last searching is performed.

☞***int LinkPtrList::indexValue(const char *value, int caseflag=0, int start=0);***

Description:

This function searches for a **leaf string** type *Link* class whose value (*Link::_val*) is the same as the first argument: *value*. Then, it returns the index (the position in the list where the first element's index is 0) of the *Link* class in the list. The second argument *caseflag* is a string

comparison option. If the value of the argument is 1, it performs case-sensitive string comparison. The last argument is the position in the list from which searching starts or to which the last searching is performed.

☞***void LinkPtrList::recurCall(void (*func)(Link *l));***

Description:

This function calls user defined function *func* such that each *Link* classes belonging to *this* class is transferred as the argument to *func*. This function can be used to modify or perform anything for each *Link* classes belonging to *LinkPtrList* class.

☞***int LinkPtrList::print(FILE *fp, int indent=0, int fmt=0);***

☞***void LinkPtrList::dump();***

Description:

These functions print out contents in *this* class. It can be used for debugging.

☞***void LinkPtrList::add(const char *fmt, ...);***

Description:

This function first build a *Link* class from an input string in *fmt* and append the built class to the list. The argument *fmt* is described in the same way as in well-known library function *printf()* and *scanf()*.

☞***Link* LinkPtrList::sub(char *hdr, int caseflag=0);***

Description:

This function subtracts a **list** type *Link* class from the list where the **header** of the *Link* class is the same as *hdr*. The second argument is a string comparison option where *caseflag=1* indicates the function performs case-sensitive string comparison. When a matching *Link* is found, it returns the pointer to the matched (and subtracted) *Link* class.

☞***Link* LinkPtrList::getNext(Link *n);***

Description:

This function returns the pointer to the *Link* class which is next to the *Link* class *n*.

4.14 CDFG

Member Data:

☞***int CDFG::_kind;***

Description:

This variable indicates the kind of the *CDFG* class, where CDFG kind is defined as

```
enum CDFG_KIND {
        C_NORMAL=1,             // Normal CDFG
        C_HYPER=2,              // edge is converted to hyper edge (multiple successors)
        C_FLAT_MOD=4,           // Loop conditional predicate is flattened into loop body
        C_FLAT_LOOP=8,    // Module class is flattened – not yet supported
};
```

Member Functions:

☞***void CDFG::convHyper(Subgraph *subg);***

Description:

This function is internally called in *CDFG::convGraph()*. It converts each edge in the graph in the form of hyper edge. Hyper edge is different from normal edge in that it has one or more successors.

☞***void CDFG::convFlatLoop(Subgraph *subg);***

Description:

This function is internally called in *CDFG::convGraph()*. It flattens the *Subgraph* of conditional predicate (the *Subgraph* returned by *Node::getCond()*) into the loop body. Refer to *diffeq.vhd* file for details.

☞***int CDFG::getKind();***

Description:

This function returns the kind of the *CDFG* class.

☞***int CDFG::convGraph(int kind, int option=0);***

Description:

This function converts the graph according to the first argument *kind*. The argument can be one of values defined in *CDFG_KIND* enumeration type. The second argument is reserved for the future use.

☞***void CDFG::print(int level=1, FILE *fp=stdout, int indent=0);***

Description:

This function prints out the contents in the graph. The meaning of arguments is the same as in *Node::print*().

☞***void CDFG::dump(const char *fname=0, int level=1);***

Description:

This function print outs contents in the graph to the file whose name is *fname*. The second argument *level* has the same meaning as in *Node::print()*.

☞***virtual void parse(const char *fname);***

Description:

This function builds internal data structure from an input file of *fname*.

☞***virtual Edge* newEdge();***

☞***virtual Node* newNode();***

☞***virtual Module* newModule();***

☞***virtual Condition* newCondition();***

Description:

These functions are used to allocate base classes (*Edge*, *Node*, *Module*, and *Condition*). They are called while parsing an input CDFG file. They are defined as *virtual* such that user can derive their own basic classes from the existing basic classes without modifying the parsing routine. For details, refer to section 4.15.

☞***virtual void parseInit(Node *n);***

☞***virtual void parseInit(Edge *e);***

Description:

These function are called during the parsing of an input CDFG file. User can redefine these virtual functions by deriving CDFG class to initialize user's own data structure during input parsing procedure. For details, refer to section 4.15.

4.15 Adding User Defined Information To The Graph

While using CDFG parser, you may want to add your own information to each node, edge, etc. The easiest way is to modify the definition of *Node* and *Edge* classes by editing **cdfg.h**. It is very error-prone and out of the concept of OOP. Moreover, you should recompile whole the source code every time you modify **cdfg.h,** which is very tedious. In this section we explain two ways how to add user defined information to the graph without modifying the parser routine.

The first method is to derive your own basic classes from the existing basic classes. For example, if you want to add an integer type variable named **bind** to each node, you can derive your own class named *MyNode* from the basic class *Node* like this.

```
class MyNode: public Node {
public:
        int bind;               // your own data structure
public:
        ...
};

// This routine shows how to access your data
void test() {
        MyCDFG graph("test.cdfg");
        Subgraph *subg = graph.getSubg();
        NodePtrList &nlist = *subg->getNodes();
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                MyNode *m = (MyNode*)nlist(pi);
                printf("%d\n", m->bind);
        }
}
```

However, the parser routine does not know that you derive your own basic class. You should also re-define the function *virtual void MyCDFG::newNode*() as show in the following example.

```
class MyCDFG: public CDFG {
public:
   virtual void newNode()        { return new MyNode; }
   virtual void parseInit(Node *n);
};
```

You can set the newly added bind variable during input parsing procedure. Assuming that the bind information is attached in **attribute** field as shown in the following example, you can properly set the your own variable by re-defining *virtual void MyCDFG::parseInit(Node *n).*

```
* This is a short segment of an input CDFG file, where bind information is attached in attribute field
node 1 + - - - (bind 1)

// This is a part of an C++ routine
void MyCDFG::parseInit(Node *n) {
  Link *bind_attrib = n->findAttrib("bind");
  if (!bind_attrib)    return;                    // error
  MyNode *m = (MyNode*)n;
   m->bind = bind_attrib->item(1)->intValue();
}
```

You can also add your own data to the *Edge* class in the same way. In this case, you should re-define *virtual void MyCDFG::newEdge()* and *virtual void MyCDFG::parseInit(Edge *e)*.

The second method is to attach your own data structure to a *Node* and *Edge* classes by using *Node::info* and *Edge::info* variable. For the same example, you can do the same job in the following way.

```
class MyCDFG::public CDFG {
public:
        virtual void parseInit(Node *n);
};

void MyCDFG::parseInit(Node *n) {
        Link *bind_attrib = n->findAttrib("bind");
        if (!bind_attrib)       return;                 // error
        int *bind = new int;
        *bind = bind_attrib->item(1)->intValue();
        n->setInfo(bind);
}

void test() {
        MyCDFG graph("test.cdfg");
        Subgraph *subg = graph.getSubg();
        NodePtrList &nlist = *subg->getNodes();
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                Node *n = nlist(pi);
                int bind = *(int*)n->getInfo();
                printf("%d\n", bind);
        }
}
```

This method has problem in memory de-allocation. The allocated memory in *void MyCDFG::parseInit()* is never de-allocated. To solve this problem, we need to derive your class as in the first method for safe de-allocation like this.

```
class MyNode: public Node {
public:
        ...
        ~MyNode()           { delete (int*)getInfo(); }
}
```

The two suggested methods seem to be simple. However, there still remain problems such as correct memory de-allocation, copying class contents, and printing out class contents. The *ExtCDFG* gives an easy way to add your own data structure to the graph without considering such problem.

4.16 ExtCDFG

As explained in the previous section, ExtCDFG class gives an easy way to add your own data structure to the graph. The ExtCDFG class supports your defined data structure by way of *Node::info* and *Edge::info* variable. To add your addition data to a *Node* class or *Edge* class, you have only to describe code according to the following rules.

4.16.1 Attaching data structure to a node

0. If you don't want to add your data structure to a node, just define **NO_EXT_NODE** before including "extcdfg.h".
1. Define your data structure name as **ExtNodeInfo** before including "extcdfg.h" file. In the case of the example in section 4.15 you can define you data structure like this.

```
struct ExtNodeInfo {
  int bind;
};
#include "extcdfg.h"
```

2. If you want to change the way that a node is printed out, describe your functionality in *ExtNode::print()* as shown in the following example. Otherwise, define **NO_EXT_NODE_PRINT** before including "extcdfg.h".

```
void ExtNode::print(int level, FILE *fp, int indent) {
    Node::print(level, fp, indent|NORET_FLAG);
    fprintf(fp, "(bind %d)\n", NODEINFO(this)->bind);
}
```

3. If you want your data to be updated during input parsing, describe *ExtCDFG::parseInit(Node *)* in the following way. Otherwise, define **NO_PARSE_INIT** before including "extcdfg.h".

4. You can get a pointer to ***ExtNodeInfo*** by using macro NODEINFO(n) as shown in the following example.

```
// print out binding info
Node *n = subg->node(1);
```

```
printf("bind = %d\n", NODEINFO(n)->bind);
```

4.16.2 Attaching data structure to an edge

0. If you don't want to add your data structure to an edge, just define **NO_EXT_EDGE** before including "extcdfg.h".

1. Define your data structure name as ***ExtEdgeInfo*** before including "extcdfg.h" file.

```
struct ExtEdgeInfo {
  int bind;
};
#include "extcdfg.h"
```

2. If you want to change the way that an edge is printed out, describe your functionality in *ExtEdge::print()* as shown in the following example. Otherwise, define **NO_EXT_EDGE_PRINT** before including "extcdfg.h".

```
void ExtEdge::print(FILE *fp, int indent) {
    Edge::print(fp, indent | NORET_FLAG);
    fprintf(fp, "(bind %d)\n", EDGEINFO(this)->bind);
}
```

3. If you want your data to be updated during input parsing, describe *ExtCDFG::parseInit(Edge *)* in the following way. Otherwise, define **NO_PARSE_INIT** before including "extcdfg.h".

4. You can get a pointer to ***ExtEdgeInfo*** by using macro EDGEINFO(e) as shown in the following example.

```
// print out binding info
Edge *e = ...;
printf("bind = %d\n", EDGEINFO(e)->bind);
```

4.17 Example

4.17.1 Node traversal

Following example routine shows how to traverse the whole graph and perform proper action for each node or edge. Since the CDFG class has hierarchical structure, you'd better perform a job for each sub-graph and perform recursion for the child sub-graphs.

```
// This example traverses all the Subgraph in the graph and prints out the contents
```

```
// The functionality is the similar to CDFG::print()
void test(Subgraph *subg) {
        NodePtrList &nlist = *subg->getNodes();
        EdgePtrList &elist = *subg->getEdges();

        // first print out edge-list
        for (Pix pi=elist.first(); pi; elist.next(pi)) {
                Edge *e = elist(pi);
                e->dump();
        }
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                Node *n = nlist(pi);
                n->dump(0);
        }

        // perform recursion for child subgraphs
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                Node *n = nlist(pi);
                // you have two ways for scanning all the subgraphs
                // The first method
                switch (n->getType()) {
                        case N_COND:
                        {
                                test(n->getCond());
                                int num_subg = n->getNumSubg();
                                for (int i=0; i < num_subg; i++) {
                                        test(n->getSubg(i));
                                }
                                break;
                        }
                        case N_MOD:
                                test(n->getSubg());
                                break;
                        case N_ITER:
                                test(n->getCond());
                                test(n->getSubg());
                                break;
                }

                // The second method
                if (n->getCond())       test(n->getCond());
                int num_subg = n->getNumSubg();
                for (int i= -1; i < num_subg; i++) {            // note that index start from –1 (refer to Cond::getSubg());
                        if (n->getSubg(i))      test(n->getSubg(i));
                }
        }
}

void main()
{
        CDFG graph("test.cdfg");
        test(graph.getSubg());
}
```

4.17.2 ASAP scheduling

```
/*****************************************************************************
*    Title : asap.cc
*    Desc  : ASAP example routine
*    Author: Jinhwan Jeon
*    Date  : 1999.12.18
*****************************************************************************/
```

```
#define      NO_EXT_EDGE
#define NO_EXT_SUBG
#define NO_PARSE_INIT
#define NO_EXT_NODE_PRINT

struct ExtNodeInfo {
        int ts;
};

#include "extcdfg.h"

#define             CHKMARK(n, m)     ((n)->getMark() & (m))
#define             SETMARK(n, m)     (n)->setMark((n)->getMark() | (m))
#define             CLRMARK(n, m)     (n)->setMark((n)->getMark() & ~(m))

void UpdateList(NodePtrList &list, Node *n, int mark)
{
    EdgePtrList &succs = *n->getSuccs();
    for (Pix pi=succs.first(); pi; succs.next(pi)) {
        Edge *se = succs(pi);
        Node *sn = (Node*)se->getSucc();

        EdgePtrList &preds = *sn->getPreds();
        int ready_flag = 1;
        if (mark && CHKMARK(sn, mark))          continue;
        for (Pix pj=preds.first(); pj; preds.next(pj)) {
            Node *pn = (Node*)preds(pj)->getPred();
            if (mark && CHKMARK(pn, mark) == 0) {
                ready_flag =0;
                break;
            }

        }
        if (ready_flag && !list.isIn(sn)) {
            list.append(sn);
                }
        }
}

void ASAP(Subgraph *subg, int ts)
{
        if (!subg)    return;
        NodePtrList &nlist = *subg->getNodes();
        Node *source = nlist.node(0);
        NodePtrList stk;

        // init schedule info
        for (Pix pi=nlist.first(); pi; nlist.next(pi)) {
                Node *cur = nlist(pi);
                cur->setMark(0);
                cur->setHwSpeed(1);                 // hardware-speed
        }

        SETMARK(source, 1);
        NODEINFO(source)->ts = ts;

        UpdateList(stk, source, 1);

        while (!stk.empty()) {
                Node *cur = stk.remove_front();

                int tend = ts;
                EdgePtrList &preds = *cur->getPreds();
                for (Pix pi=preds.first(); pi; preds.next(pi)) {
```

```
                Node *pn = preds(pi)->getPred();
                int tend_pn = NODEINFO(pn)->ts + pn->getHwSpeed();
                if (tend_pn > tend)             tend = tend_pn;
        }
        NODEINFO(cur)->ts = tend;

        switch (cur->getType()) {
                case N_MOD:
                {
                        ASAP(cur->getSubg(), tend);
                        int delay = NODEINFO(cur->getSubg()->node(-1))->ts - tend;
                        cur->setHwSp eed(delay);
                        break;
                }
                case N_ITER:
                case N_COND:
                {
                        ASAP(cur->getCond(), tend);
                        int delay1 = NODEINFO(cur->getCond()->node(-1))->ts - tend;
                        int delay2 = 0;
                        for (int i=cur->getNumSubg()-1; i>=-1; i--) {
                                Subgraph *subgi = cur->getSubg(i);
                                if (!subgi)   continue;
                                ASAP(subgi, tend+delay1);
                                int d = NODEINFO(subgi->node(-1))->ts - tend - delay1;
                                if (d > delay2)                 delay2 = d;
                        }
                        cur->setHwSpeed(delay1+delay2);
                        break;
                }
        }
        SETMARK(cur, 1);
        UpdateList(stk, cur, 1);
    }
}

void main(int argc, char **argv)
{
    if (argc < 2)       return;
    ExtCDFG graph;
    graph.parse(argv[1]);

    ASAP(graph.getSubg(), 0);
}
```

4.18 Compile and Link

When compiling your program with our CDFG tool, you should include "cdfg.h" in your source code where the file is located in $CDFG_HOME/include directory. You should add the path as an include path by using –I option of C-complier. When linking, you should link your programs with $CDFG_HOME/lib/cdfg.a.

5. CDFG Viewer

CDFG viewer is activated executing ***cdfgview.*** It parses a CDFG file selected by the user and draws a graph on a display window. It shows graphically the CDFG as well as the HLS information such as scheduling and binding.

5.1 Simple CDFG

A simple CDFG, is a CDFG with no information about HLS such as scheduling and binding. It is a form of CDFG before HLS. Figure 5.1 shows a simple CDFG for the example of ***biquad***.

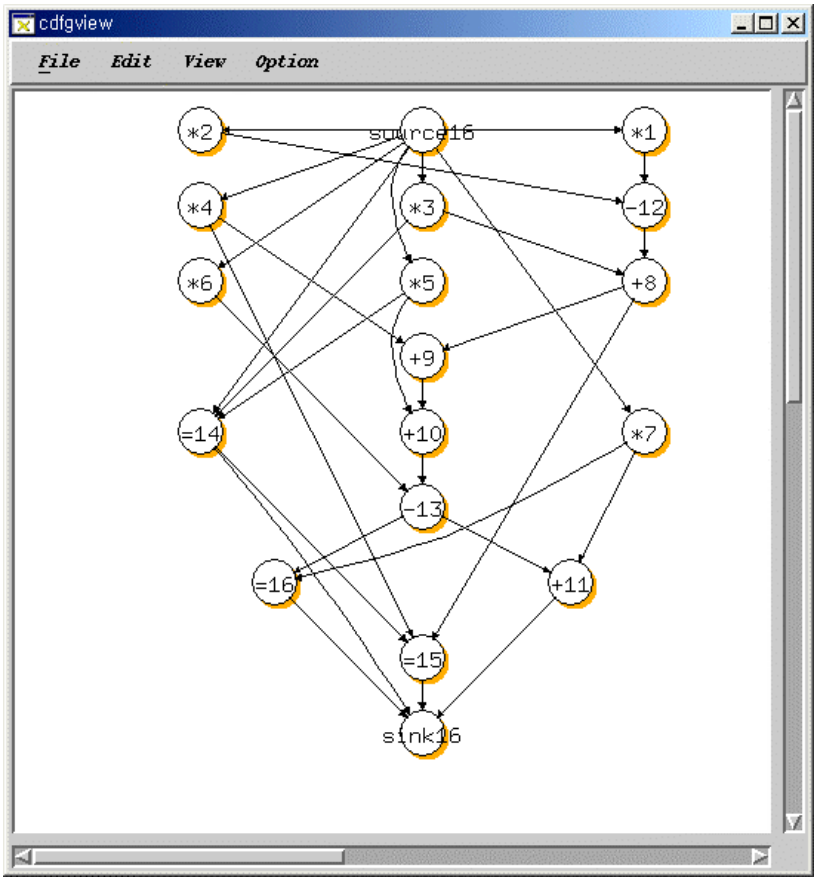


Figure 5.1. An example of simple CDFG.

5.2 Annotated CDFG

An annotated CDFG, is a CDFG with annotated information on HLS. The information includes

scheduled control steps (CSTEPs) for the operations and binding of resources such as FUs and registers.

5.2.1 Control step

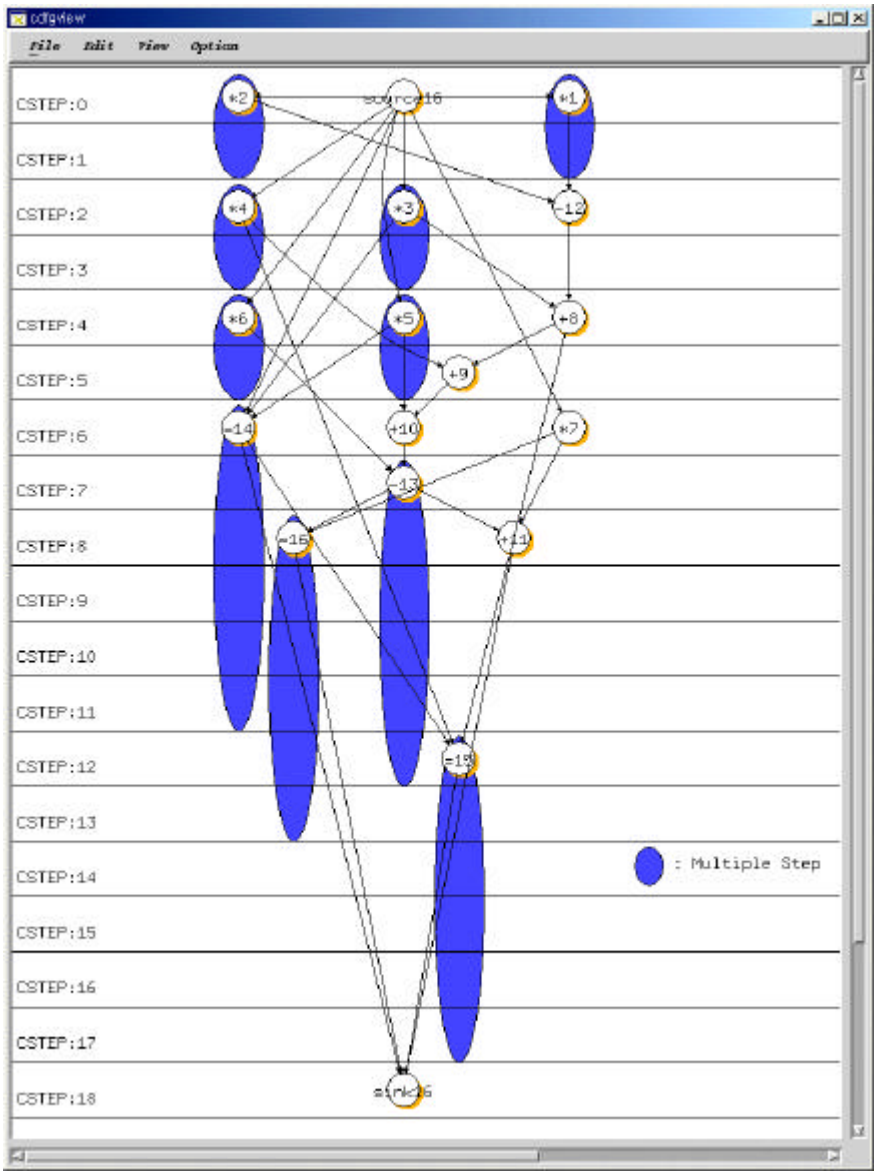


Figure 5. 2. An example of CDFG including scheduling information.

Figure 5.2 shows the CSTEP where each operation is scheduled. It also shows the spans of multi-cycle operations. The control step information is displayed by pushing the button ***Cstep*** in the ***View*** menu.

5.2.2 Allocation of registers

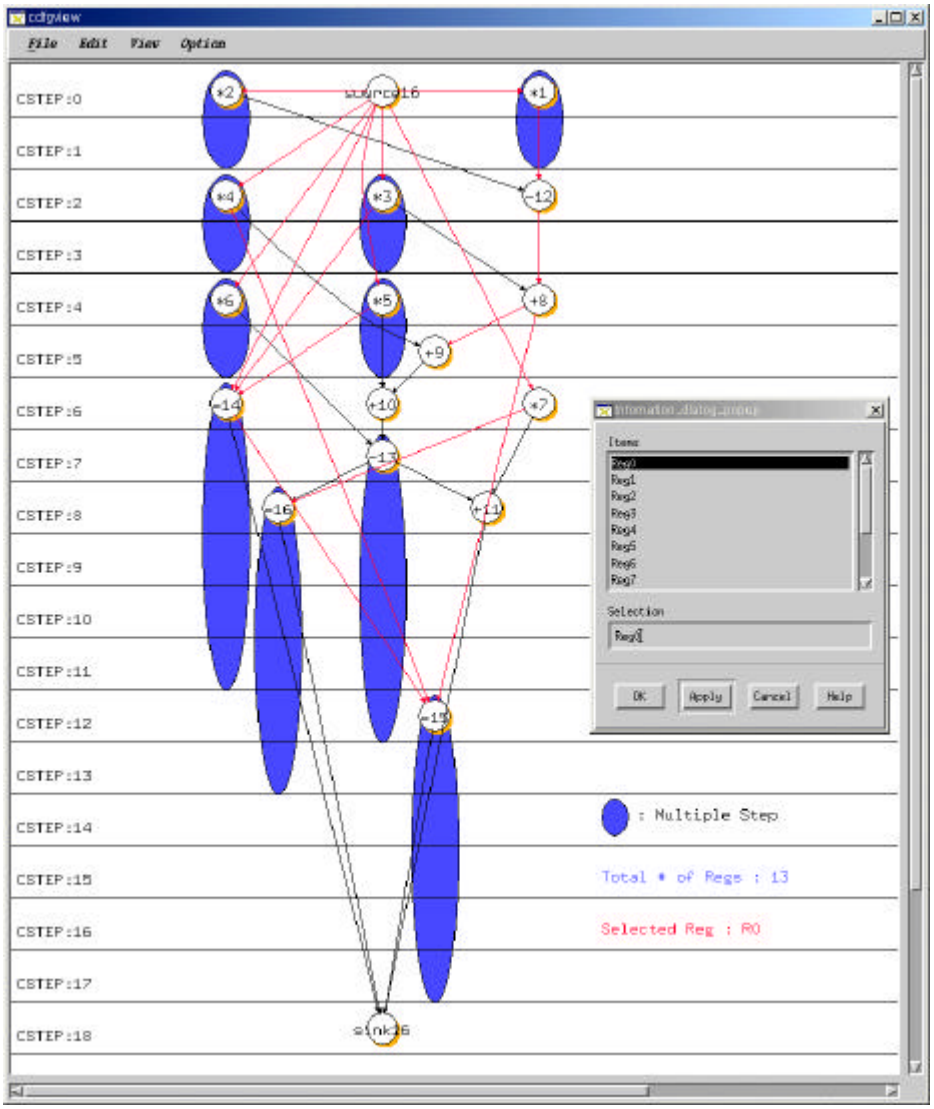


Figure 5. 3. An example of CDFG including register binding information.

Figure 5.3 shows register binding information. We can select a register to see which edges are bound to it. In this example, total number of registers is 13 and the selected register is *R0*. When we push the *Apply* button in the dialog box, the edge lines bound to the selected register become red. In this way, the user can check which edges are bound to a register. The dialog box is activated with the button ***Reg_alloc*** in the ***View*** menu.

5.2.3 Binding of functional units

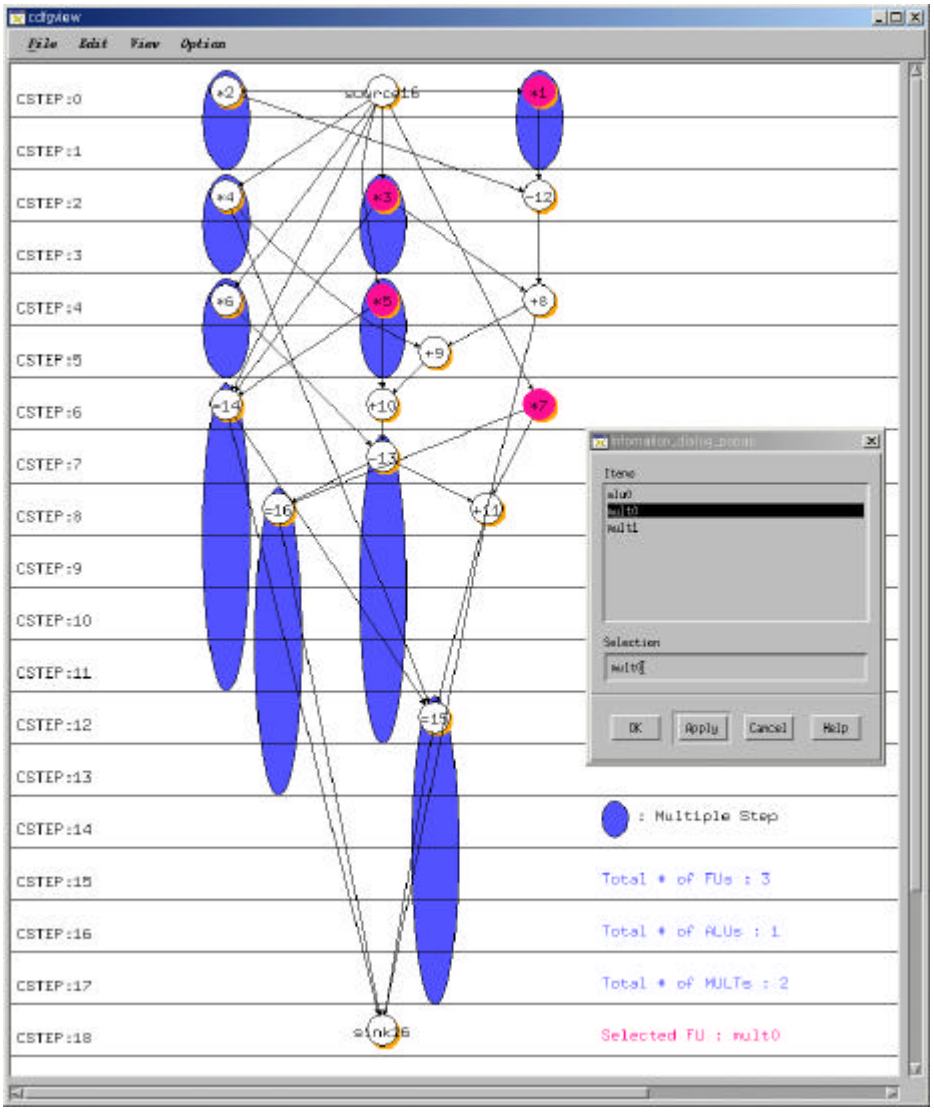


Figure 5. 4. An example of CDFG including FU binding information.

Figure 5.4 shows functional unit binding. We can select a functional unit to see which vertices are bound to it. In this example, total number of functional units is 3 including one ALU and two Multipliers and *mult0* is selected. When we push the *Apply* button in the dialog box, the vertices bound to the selected register become red. In this way, the user can check to see which vertices are bound to a functional unit. The dialog box is activated with the button ***Binding*** in ***View*** menu.

References

[1] S. Park, K. Choi, Design Automation Lab, Seoul National Univ. *VHDL Developer's Toolkit 2.6 User's Guide & Reference* Technical Report No. SNU-EE-TR-1997-5.

[2] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGRAW-HILL international editions, 1994.

e-mail : jeonjinh@gctsemi.com, ayjin@poppy.snu.ac.kr