GRAPHGEN

Control/Data Flow Graph Generator
For Full-VHDL

version 1.4

LEDA S.A.

35 Avenue du Granier
38240 Meylan, France

Tel: (+33) (0)4 76 41 92 43
Fax: (+33) (0)4 76 41 92 44
E-mail: sales@leda.fr, support@leda.fr

GRAPHGEN

Control/Data Flow Graph Generator for Full VHDL

LEDA S.A. 35 Avenue du Granier 38240 Meylan France
Tel: (+33) (0)4 76 41 92 43 — Fax: (+33) (0)4 76 41 92 44

Implementor's Guide

Version 1.4

TABLE OF CONTENTS

1 . INTRODUCTION

GRAPHGEN is a graphical representation for the synthesis of full-VHDL (both VHDL'87 and VHDL'93). This means that all the benefits of VHDL can be used for describing hardware. The number of different nodes necessary is not significantly different from any other graphical representation for synthesis.

The factor that enables such a set of graphs to be generated is the use of **LVS**, a full VHDL 87/93 compiler that generates an intermediate representation from which the graphs and other synthesis-related information are extracted.

GRAPHGEN is a tool for the generation of Control Flow Graphs (CFG) and Data Flow Graphs (DFG) from full-VHDL ('87 and '93). The graphs are intended to be used by behavioral and logic-level synthesis tools. **GRAPHGEN** can also partition the graphs into basic block and execution path structures enabling the application of some of the more common high-level synthesis transformations.

GRAPHGEN is part of the **LVS** toolbox. It can be used alone or with other **LVS** tools such as **APEX** or **GEME**. By using these tools together with **GRAPHGEN**, the graphs generated can be optimized even further (see § 4.2). For further information about other tools in the **LVS** toolbox, contact sales@leda.fr.

IMPORTANT

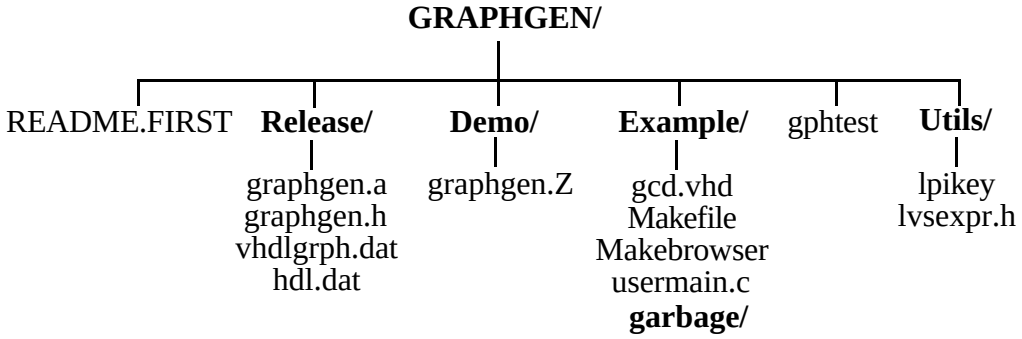
GRAPHGEN is not a standalone tool. It is part of the LEDA VHDL System (LVS) toolbox. Therefore, **GRAPHGEN** must be used with the LVS compiler. This has the benefit of enabling the user of **GRAPHGEN** to take advantage of all of the facilities of LVS: accessible and expandable intermediate format, procedural interface to this format, browser and so on. This version of **GRAPHGEN** must be used with versions after 4.1 of the LVS compiler.

GRAPHGEN can also be used with other elements of the LVS toolbox such as APEX and GEME.

This document only describes the definitions in LVS relevant to **GRAPHGEN**. These definitions include new intermediate nodes and primitives. All applicable routines in the original LVS intermediate format are still valid (for example, vifOpenUnit, vifCreateNode,…). For more details on LVS and its environment, the user is referred to the LVS Implementor's Guide.

2. INSTALLATION AND TEST

The installation diskette for **GRAPHGEN** contains the following files and directories:



File **README.FIRST** explains how to install **GRAPHGEN** once it has been copied to the host machine;

Directory **Release** contains the following files:

graphgen.a: **GRAPHGEN** archive to be linked with applications;
graphgen.h: Header file to be included with all applications using **GRAPHGEN**;
vhdlgrph.dat: **GRAPHGEN** intermediate format. Used by **schemagen** to extend the LVS schema to include **GRAPHGEN** nodes, classes and attributes.
vhdl.dat: Include file for **schemagen**. All files representing the LVS schema and extended schema must be referred to here. For **GRAPHGEN** only (no user-extension, the file contains two lines:

```
#include vhdlleda.dat
#include vhdlgrph.dat
```

vhdlleda.dat is the LVS schema and **vhdlgrph.dat** is the **GRAPHGEN** schema extension.

To build other extensions, this file must be modified to refer to files containing other extensions. For example:

```
#include vhdlleda.dat
#include vhdlgrph.dat
#include vhdluser.dat
```

vhdluser.dat is a user schema extension.

For more information, please refer to the LVS Implementor's Guide, Part I, Chapter "Schema Generator".

Demo is a directory containing a **GRAPHGEN** executable in compressed format. This allows **GRAPHGEN** to be executed in standalone mode.

Example is a directory containing the following:

usermain.c:	A sample C file showing how **GRAPHGEN** may be invoked from any application.
gcd.vhd:	A VHDL file used to test **GRAPHGEN** during installation.
Makefile:	A make file linking an application to LVS (uvif.a, lvskernel.a) and **GRAPHGEN** archive. This file is executed by gphtest.
Makebrowser:	A make file linking the newly created VIF (uvif.a) to LVS archives lvskernel.a and lvsbrowser.a. This generates an extended browser allowing the VIF extensions created by **GRAPHGEN** to be viewed normally. The executable generated is called "browser" (see LVS Implementor's Guide, Part I, Chapter "Schema Generator").
garbage/:	Temporary directory used to store object files.

gphtest is a script file that can be executed to install and test **GRAPHGEN** (see §2.2).

Utils is a directory containing the following:

lpikey:	Executable for passing authorization key to application.
lvsexpr.h:	Header file containing useful functions that may be called from an application. If any of these functions is to be used, this file must be included in the application.

2.1 Installing GRAPHGEN

To install **GRAPHGEN**, change directory to where you want it to reside and execute the command

tar xvpf /dev/fd0

Then, perform the following steps:

1. Install LVS

LVS is LEDA's VHDL System, containing a full VHDL'93 / VHDL'87 compiler and a procedural interface allowing access to an intermediate representation of the compiled VHDL model. **GRAPHGEN** uses and builds upon this intermediate format.

For information on how to install LVS, please refer to the "LEDA VHDL SYSTEM - Implementor's Guide Part 1", Chapter 2.

IT IS ASSUMED THAT AT LEAST LVS VERSION 4.1 IS AVAILABLE

2. Set Environment Variables

It is recommended to set and use the environment variable LVS_PATH to indicate the physical location of LVS. This is the directory that corresponds to "install_dir" when following the LVS installation instructions (see step 1). This can be done by executing:

setenv LVS_PATH /usr/local/lvs

to indicate the installation directory of LVS. This directory will hereafter be called **$LVS_PATH**.

3. Validate GRAPHGEN

The **GRAPHGEN** package has two usable files: **graphgen.a** in the **Release** directory is an archive that can be linked with any user application and **graphgen** in the **Demo** directory is an executable meaning that **GRAPHGEN** can be used in standalone mode. To use one or both of these files, they have to be validated with an authorization key supplied by LEDA (support@leda.fr).

Validation is done with **lpikey** in directory **Utils** of the **GRAPHGEN** diskette.

3.1 Validating archive

The LVS archive **lvskernel.a** must be updated with the authorization key for **GRAPHGEN**. This is done by changing directory to **$LVS_PATH/lvskern** and executing:

lpikey <KEY>

where <KEY> is **GRAPHGEN**'s authorization key.

3.2 Validating executable

The executable is validated directly.To do this, change directory to where the executable is located (default: Demo) and type:

uncompress graphgen.Z
lpikey -f graphgen <KEY>

For more information on **lpikey**, please refer to the Appendix A of this document.

2.2 Testing GraphGen

In this section we show how to test both **graphgen.a** and the executable **graphgen**.

2.2.1 Testing Archive

In the **GRAPHGEN** package, there is a script named **gphtest** that can be used not only to test the **graphgen.a** archive but also serves as an example of how to build applications that use **GRAPHGEN**.

To run the script, simply type:

gphtest <KEY>

where <KEY> is the authorization key for **GRAPHGEN**.

This has the effect of creating and validating a new archive, **uvif.a** which contains the extended intermediate format. This archive is created in the **Example** directory.

The archive will be used when **gphtest** tests the installation. This is done by building a new executable called **mygph** in the **Example** directory. The executable consists of one compiled C file, **usermain.c**, linked with **uvif.a**, **lvskernel.a** and **graphgen.a**. A **Makefile** is supplied in the Example directory to create **mygph**. **GRAPHGEN** is invoked in usermain.c through the inclusion of the **graphgen.h** header file and by calling the function *gphGraphGen()* which is part of **GRAPHGEN**'s procedural interface (see section 4).

NOTE: The order in which these archives are linked is very important.

Also in the **Example** directory, a VHDL file called **gcd.vhd** is supplied. This file will be compiled into a VHDL library, **LIB**, and **mygph** will be executed on the architecture *BEHAVIOR/GCD* to extract the basic blocks and execution paths from the process contained in the architecture. At the end of this execution, **mygph** should have created 7 (0 to 6) basic blocks and 3 paths. The information shown below should be printed on the screen. If this is not the case, please contact support@leda.fr.

NOTE: **gphtest** creates (and subsequently destroys) many files and directories. If, for any reason, this script is interrupted, these files may be removed by executing the last three lines of gphtest. If you want to save the created environment, simply comment out these three lines.


```
For Process GCD :
================================
BLOCK 0
      node cfg_general_operation      12
      node cfg_general_operation      13
     node cfg_wait   14
Successors:
     Block 1
BLOCK 1
      node cfg_general_operation      15
      node cfg_general_operation      16
Successors:
     Block 2
BLOCK 2
     node cfg_loop   17
Successors:
     Block 3
     Block 6
BLOCK 3
      node cfg_boolean_branch
Successors:
     Block 4
     Block 5
BLOCK 4
      node cfg_general_operation      19
Successors:
     Block 2
BLOCK 5
      node cfg_general_operation      21
Successors:
     Block 2
BLOCK 6
      node cfg_general_operation      24
Successors:
```

```
     Block 0

For Process GCD :
================================
PATH 1
      node cfg_general_operation      12
      node cfg_general_operation      13
     node cfg_wait   14
      node cfg_general_operation      15
      node cfg_general_operation      16
     node cfg_loop   17
      node cfg_boolean_branch
      node cfg_general_operation      19
SUCCESSOR cfg_loop      17
PATH 2
      node cfg_general_operation      12
      node cfg_general_operation      13
     node cfg_wait   14
      node cfg_general_operation      15
      node cfg_general_operation      16
     node cfg_loop   17
      node cfg_boolean_branch
      node cfg_general_operation      21
SUCCESSOR cfg_loop      17
PATH 3
      node cfg_general_operation      12
      node cfg_general_operation      13
     node cfg_wait   14
      node cfg_general_operation      15
      node cfg_general_operation      16
     node cfg_loop   17
      node cfg_general_operation      24
SUCCESSOR  cfg_general_operation 12
```

2.2.2 Testing Executable

Before testing the executable, it must have been validated with **lpikey**. Make sure that step 3.2 in section 2.1 was carried out.

Change directory to Demo. Ensure that the executable for the LVS compiler ("v") is visible. Then, type the following sequence of commands:
To enter the LVS environment, simply type:

$LVS_PATH/v

The prompt will change to **LVS>**

To create a new working library "LIB", type

LVS> new LIB

To create libraries STD and IEEE, type

LVS> createstd

to make theses libraries visible, type

LVS> add STD

LVS> add IEEE

to compile **gcd.vhd** in the Example directory, type

LVS> comp ../Example/gcd.vhd

This will create two library units in the library LIB: enity GCD and architecture BEHAVIOR/GCD.

Finally, quit the LVS environment by typing

LVS> quit

There are several parameters that can be put on the graphgen command line. These are outlined in section 2.3. To obtain the same results as in section 2.2.1, we type the following:

graphgen LIB BEHAVIOR/GCD -ps -b -d

If there is some discrepency between the results obtained here and in section 2.2.1, please contact LEDA (support@leda.fr).

2.3 Command Line Options

The **GRAPHGEN** executable can accept several parameters on the command line. These correspond more or less to the actual parameters that can be passed when using graphgen.a and calling **GRAPHGEN** from a user-application. In standard operation, **GRAPHGEN** generates a control flow graph (CFG) for the process equivalent of each concurrent statement, as well as for all subprogram bodies in the input description. For each CFG node, a data flow graph (DFG) may be generated. If it is preferable to attach the graphs to the concurrent statement nodes rather than to the equivalent process, the **-n** option must be used in the executable and the **CreateEquivalentProcess** parameter must be set to False in the procedural interface. Graph information may be attached to different concurrent statement nodes (or their equivalent process) in the unit processed.

In the section on testing **GRAPHGEN**, we saw how to execute **GRAPHGEN** to simply generate CFGs and DFGs. **GRAPHGEN** is also capable of partitioning CFGs and merging DFGs. The two partitioning mechanisms available are **basic blocks** and **execution paths**.

If the user wants **GRAPHGEN** to generate basic block partitions, **GRAPHGEN** must be executed with the "**-b**" argument. For example,

graphgen -b

which will prompt for the library and unit names, or

graphgen -b LIB behavior/gcd

which executes directly.

If execution paths are required, we can use the **-p** or **-po** option to generate optimised paths or the **-ps** option to generate simple paths (see §3.4.2).

graphgen -ps LIB behavior/gcd

If both basic blocks and execution paths are required, both parameters should be entered on the command line. For example:

graphgen -b -po LIB behavior/gcd

In general, conditional expressions of equality or inequality with the RHS a locally static value are automatically assumed to be executed in the controller and the *dfg_model* attribute of the corresponding cfg node is always NULL. However, an option exists, **-d**, which allows the user to decide where such expressions should be evaluated. Consider the following code segment:

if COND='1' **then** …

If the **-d** option does not appear on the command line, this statement generates a **cfg_boolean_branch** node with the *dfg_model* attribute empty. The expression itself can be found by following the *cfg_vif_model* attribute.If the **-d** option does appear on the command line, the *dfg_model* for the above example will be filled.

Other options are:

-f Force regeneration of graphs regardless of whether they have already been generated.

-n Do **not** transform concurrent statements into their equivalent processes before execution.

3 . GRAPHGEN: RATIONALE

In this section we present the main principles and motivations behind **GRAPHGEN** in terms of the graphs themselves. Throughout this section, we will use the VHDL example of figure 3 to illustrate the different concepts behind **GRAPHGEN**

Almost all synthesis tools start with some form of graphical representation that models the control flow and/or the data flow of the original description in a hardware-oriented manner. In general, each statement in the input description maps directly onto one or more nodes in the graph domain. Synthesis then consists of applying transformation algorithms to the graph domain in order to generate a netlist of hardware components and their interconnections.

The intermediate graphical representation tends not to receive as much attention as it deserves, mainly because synthesis tool developers prefer to concentrate on the classical synthesis algorithms such as scheduling and allocation. For example, many synthesis tools claim to start with standard hardware description languages (such as VHDL) as their input whereas, in reality, subsets of these languages are used.

The use of subsets is not only due to the limitation of the synthesis algorithms and/or the irrelevance of some statements to hardware, but also due to the unavailability of front-end tools (compilers generating intermediate formats, graph generation tools, etc.). If a VHDL statement cannot be mapped directly onto an available graphical template, it is usually quite simply rejected regardless of the benefits it may provide. This is unfortunate as languages such as VHDL contain very powerful constructs, particularly for data typing and synchronisation, that can considerably ease the design of hardware. Rejecting such constructs means that designers have to find roundabout ways of describing equivalent statements thereby increasing the size of the description and hence the design time and maintenance costs.

GRAPHGEN allows all VHDL constructs to be mapped onto the graphical domain for synthesis. **GRAPHGEN**'s model consists of a control flow graph (CFG) for each concurrent statement in a VHDL input description. For each CFG node, a data flow graph (DFG) may be created. CFG nodes may be grouped into control blocks or control paths and DFGs may be subsequently merged for optimisation purposes. In this section we will show the nodes available for representing full-VHDL in the graph domain.

3.1 Control Flow Graph Nodes

Control flow graphs are used to represent the different possible sequences of execution for each set of statements in a VHDL process. From a hardware point of view this sequentiality is not always necessary as some mutually exclusive statements can be executed in parallel if the appropriate resources are available. It is the task of the synthesis algorithms, in particular the scheduling algorithm, to identify and to group such statements into what are known as *control states*.

A control flow graph (CFG) is defined as:

$$CFG=(Vc,Ec)$$

where *Vc* is a set of nodes corresponding to different VHDL statements:

$$Vc = Vw \cup Vb \cup Vl \cup Ve$$

where

Vw = synchronisation nodes (wait)
Vb = branch nodes (if, case, exit, next)
Vl = loop nodes (while, loop)
Ve = other nodes (assignments, procedure calls,…)

and Ec is the set of edges identifying the flow of control

$$e = (v1, v2, c), \text{ where } v1, v2 \in Vc.$$

In other words, the edge *e* represents the fact that node *v2* is executed after *v1* if condition "c" is true. The node *v2* is said to be a successor of *v1*. Only one successor can be taken from any node. If a node has more than one successor then a condition must be evaluated to determine which one is to be taken.

Figure 1 shows the set nodes (Vc) available for building a CFG.

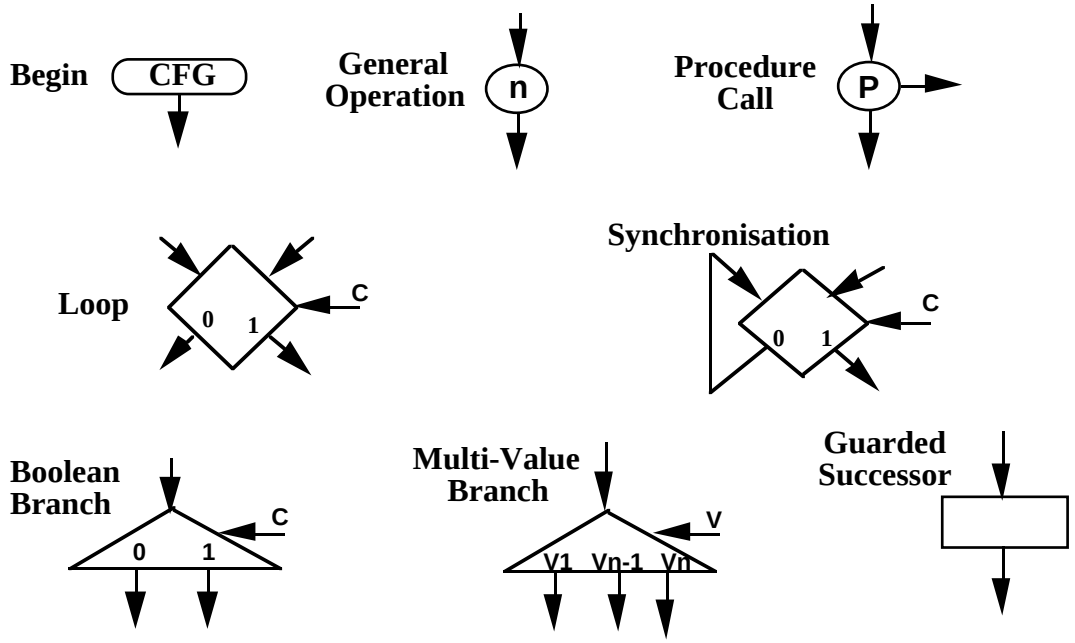


Figure 1: Set of CFG nodes

3.2 Data Flow Graph Nodes

At first, for each CFG node, an independent DFG is generated depending on whether there is any data flow in the CFG operation. These DFGs can subsequently be merged to produce a more optimised result.

VHDL permits the usual set of unary, binary, boolean and logical operations as well as function calls to evaluate expressions. With the exception of function calls, these operations can be easily represented using standard DFGs. Where VHDL is most powerful however, is in the different data types that are allowed as operands, parameters and targets of the expressions. Most synthesis tools severely limit the data types accepted and thus lose a great amount of the versatility offered by VHDL. In our graph domain, all VHDL data types can be represented.
The DFG nodes allowed are shown in figure 2. They can be divided into four sets corresponding to the hardware that will be allocated to them.

(i) **Operator nodes**
These will be allocated functional units in the synthesised model. They are used to represent unary and binary operations, functions appearing in function calls (possibly referring to predefined functional units in a specific library), and an abstract timer node that is used to model specific delays requested in the input description (**wait for** 10 **ns**; etc.). It is assumed that, if such delays are requested, then the synthesis environment has access to a functional unit capable of calculating them and which can be mapped directly onto this node.

(ii) **Operand nodes**
These nodes will eventually be allocated memory elements in the synthesised model. Different nodes are used to model memory reads and writes. The operand nodes can represent ROMs (simple constant values) or RAMs. RAMs can in fact be simple registers (operand read/write) or memory devices necessitating an addressing mechanism (array read/write).

(iii) **Edge nodes**
These will be allocated to buses or multiplexors in the synthesised model, depending on the target architecture. The nodes represent simple connections within the data-path (edge) and between the data-path and the controller (dfg/cfg interface) as well as nodes for merging buses when a VHDL aggregate is used.

(iv) **Address nodes**
The final set contains nodes that do not explicitly map onto a hardware resource but help to determine the type of resource required. These include parameter nodes and operand addressing nodes. Address nodes are used between operands and expressions that calculate the address of the operand to be accessed. This address calculation can include nodes from all sets, including address nodes.

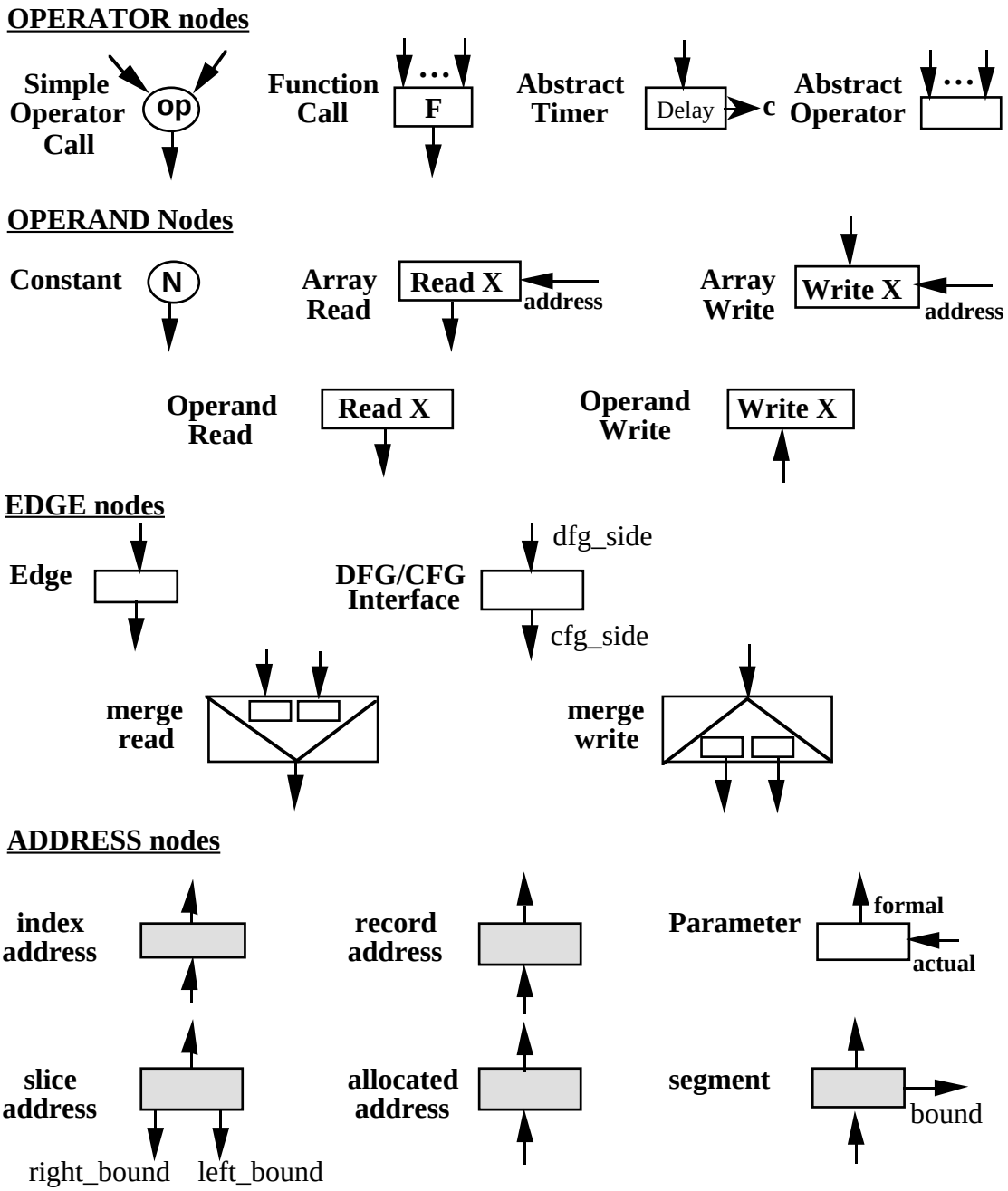


Figure 2: Set of DFG nodes

A data-flow graph (DFG) is defined as:

$$DFG = (Vd \cup Od \cup Ad, Ed)$$

where,

- Vd is the set of operator nodes
- Od is the set of operand nodes
- Ad is the set of address nodes
- Ed is the set of edges linking operators and operands.

Each DFG is built as a directed acyclic graph.

To show the generation of control and data flow graphs for various VHDL statements more clearly, we will use the VHDL code outlined in figure 3. The process describes a bubble-sort algorithm that sorts a set of records according to the date in one of the record fields. The numbers in comments will be used when referring to statements on the same line. When a rising edge appears on the "START" port (statement 1), a loop is executed in order to read in the records to be sorted (statements 3 to 8). This loop terminates when either the maximum number of records is specified, as indicated by the generic parameter "MAXSIZE" or when a special terminating record is encountered. This record is identified by the fact that the date field has all elements set to zero (statement 7).

Once all records have been read into the RAM, bubble sorting can begin. This algorithm consists of two nested loops. The inner loop compares two adjacent records. For this example, a comparison of the day of the month only is executed (statement 15). It is assumed that the month and year are equal for clarity purposes. A swop is then executed if necessary. A pointer to the RAM is incremented and the inner loop iterates. In this way a wrongly-positioned record "bubbles" its way, element by element, to the correct position. The outer loop iterates when a full pass of the RAM, less the sorted elements, has been executed by the inner loop.

The description in figure 3 makes good use of VHDL's data typing capabilities to represent the data. It also uses powerful assignment statements to modify data (statements 5, 6, 19 and 20). Most synthesis tools do not accept such descriptions and, in order to describe such an algorithm, require both more and simpler object declarations that will possibly lead to more hardware.


```
package DATATYPES is
   type DATEARRAY is array (0 to 2) of integer;
   type DATAPACKAGE is array (0 to 255) of bit;
   type INFO is record
      DATE : DATEARRAY ;                --day,month,year
      DATA : DATAPACKAGE ;              --not interested in contents
   end record;
end DATATYPES ;

use work.DATATYPES .all;
entity BUBBLE is
   generic (MAXSIZE : integer := 255);
   port( START      : in  bit;
         ACKOUT     : out bit;
         VALIDIN    : in  bit;
         DATEIN     : in  DATEARRAY ;
         DATAIN     : in  DATAPACKAGE );
end BUBBLE ;

use work.DATATYPES .all;
architecture BEHAVIOR of BUBBLE is
   type MEMORY is array (0 to MAXSIZE ) of INFO;
begin
   BUBBLESORT : process
      variable I,K,ITER,NUMELTS   : integer;
      variable TEMP1              : INFO;
      variable RAM                : MEMORY ;
   begin
      wait until START'EVENT and START = '1';          --1 Executed in CFG
      NUMELTS:= 0;                                     --2
      while (NUMELTS<= MAXSIZE ) loop                  --3
         wait until  VALIDIN = '1';                    --4
         RAM(NUMELTS) := (DATEIN,DATAIN);              --5 Assign by record agg
         ACKOUT<='1','0'  after 5 ns;                  --6 Acknowledge receipt
         exit when RAM(NUMELTS).DATE=(0,0,0);          --7 Last packet
         NUMELTS:= NUMELTS+ 1;                         --8
      end loop;
      I := 1;                                          --9
      while I <= NUMELTS loop                          --10
         ITER := NUMELTS + 1;                          --11
         while ITER > I loop                           --12
            ITER := ITER - 1;                          --13
            TEMP1 := RAM(ITER);                        --14
            if TEMP1.DATE(0) < RAM(ITER-1).DATE(0) then --15 Swop
               RAM(ITER) := RAM(ITER-1);               --16
               RAM(ITER-1) := TEMP1;                   --17
               if RAM(ITER).DATE(0)<=0 then            --18  Last Packet
                  RAM(ITER).DATE(1 to 2) := (2=>99,1=>13); --19  Assign by named agg
                  RAM(ITER).DATA := ('0',others=>'1');  --20  Assign by pos agg
               end if;
            end if;
         end loop;
         I := I + 1;                                   --21
      end loop;
   end process BUBBLESORT;
end BEHAVIOR;
```

Figure 3: VHDL description of bubble-sort algorithm

3.3 Representing Full VHDL

For each VHDL sequential statement, a CFG Node is created (this is not entirely true, as will be explained in the section on CFG optimisation). This node contains a (possibly NULL) pointer to a DFG modelling the data flow through that statement. It also contains a (possibly empty) list of pointers to other CFG nodes that are the successors of that node. The CFG for the process in figure 3 is shown in figure 4. The numbers indicate which statement of figure 3 corresponds to the CFG node. The arcs indicate the successors of a given node.

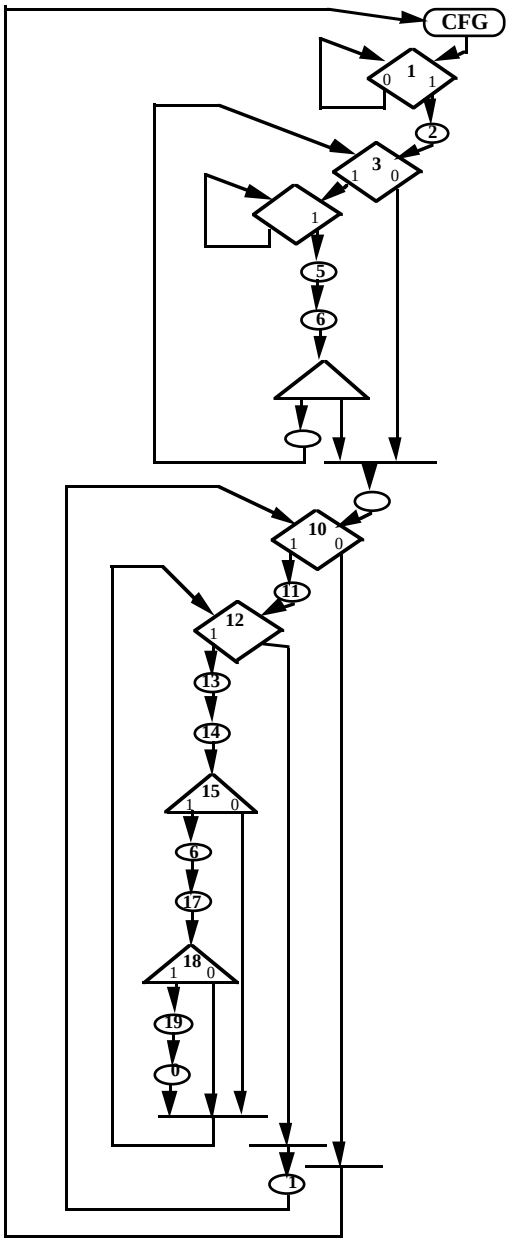


Figure 4: Control flow graph for description of figure 3

A brief discussion of the different CFG nodes and the corresponding DFGs follows:

3.3.1 Synchronisation Statements

Synchronisation in VHDL is modelled by the "wait" statement. VHDL "wait" statements are normally treated by synthesis tools as implying a change of state. In other words, statements appearing before and after a "wait" statement can never execute in the same control state (i.e. potentially in parallel). Therefore, depending on where statements are placed with respect to "wait" statements, the resulting hardware can change radically. For example, in the following code:

```
A:=B+1;
wait until CLK='1';
X:=Y-1;
```

both variable assignments can execute before or after the "wait" statement with no affect on the result. If both appear on the same side of the "wait" statement, they can be executed in parallel thereby possibly speeding up the overall execution. However, being on different sides means that they execute in different states and can thus potentially share hardware resources.

Most logic synthesis tools severely restrict the use of "wait" statements. They must normally be used with one and only one synchronisation signal per process (identified as a clock signal and having only '0' to '1' or '1' to '0' transitions). The reason for this limitation is the fact that the target architectures of such synthesis tools are completely synchronous and it must be possible to easily identify the clock.

With behavioural synthesis tools, the use of any number of "wait" statements each using complex conditions involving different signals is becoming possible. Although a "wait" statement still implies a change of state, there is no longer a need to identify a system clock.
It is very important to treat "wait" statements efficiently in the graph domain. Thus, a dedicated synchronisation node is used uniquely to model these statements. In figure 4, we see that synchronisation nodes are generated by statements 1 and 4 of figure 3. Both model the synchronisation of the process with the rising edge of an incoming signal. This is stated explicitly in statement 1 through the use of the "EVENT" attribute and implicitly in statement 4, as a "wait until" statement automatically implies that an event must occur. Thus, both synchronisation statements are modelled in exactly the same manner in the graph domain.

Neither synchronisation statement provokes the generation of a DFG. This is because the condition is a simple bit equality operation and, from the hardware point of view, it is more efficient to execute such operations in the controller. If the condition was more complex (involving types other than bits) a DFG would be generated. This DFG returns a boolean value indicating the result of the condition. This can be seen in figure 5.

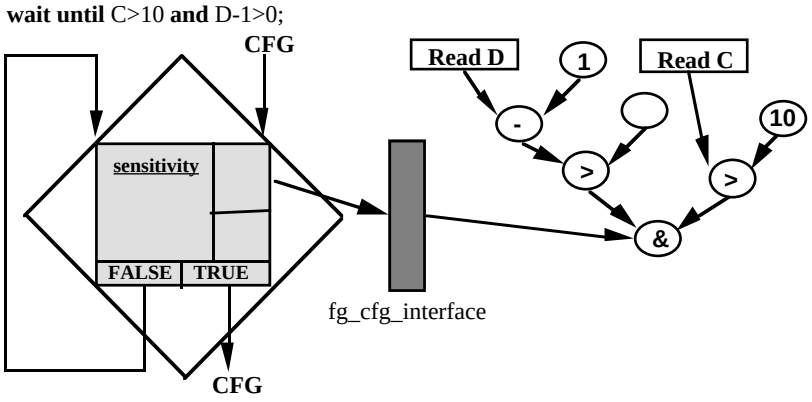


Figure 5: Wait condition executed in DFG

The "wait" condition can in fact contain three different clauses: a sensitivity clause ("wait on"), a condition clause ("until") and a timeout clause ("for"). If the sensitivity clause is present, the process waits until there is an event on one of the items of the sensitivity list. If the condition clause is also present, the process only continues if the condition evaluates to true. If the timeout clause is present, the process continues when this time has elapsed regardless of the state of the other two clauses.

Figure 6 depicts the graphs generated for a complex "wait" statement. The sensitivity clause creates no DFG, but a list of signals to which it is sensitive (along with the appropriate edges) is attached to the node. In hardware terms, the test for events on elements of this list will be performed in the controller. The condition clause and the timeout clause both create independent sub-DFGs. These sub-DFGs are independent because while the condition DFG is invoked only on events, the timeout DFG is invoked immediately, regardless of any events.

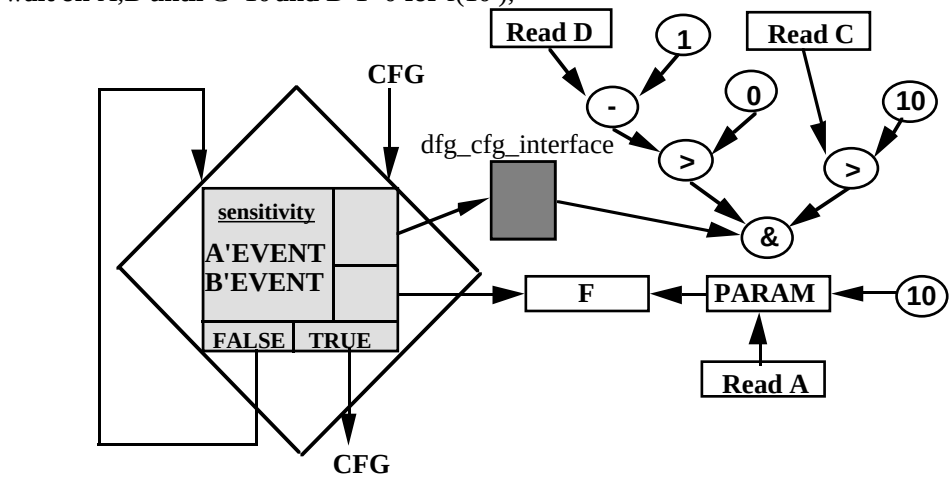


Figure 6: Modelling complex wait statements

3.3.2 Flow of Control Statements

Apart from "wait" statements, the main flow of control statements in VHDL processes include branch statements ("if", "case", "exit" and "next"), loop statements ("while", "loop", "for") and procedure calls. Other statements such as assignments generate a CFG general operation node having only one successor. The complexity of such statements is passed to the DFG.

3.3.2.1 Branch Statements

VHDL branch statements are mapped onto boolean branch nodes and multi-value branch nodes (for "case" statements only) in the CFG. Conditions are treated as for synchronisation statements. In other words, if it is a simple binary equality operation, it is assumed that it will be executed in the controller and no DFG is generated. Otherwise a DFG is generated and the result fed back via a dfg_cfg_interface node.
VHDL "next" statements are treated like "exit" and simple "if" statements in that they create a single boolean branch node. The difference between "exit" and "next" statements is their successors within a loop. For "next" statements, the true output points to the containing loop statement and the false output points to the succeeding statement. For "exit" statements, the true output points to the statement succeeding the loop region and the false output points to the

statement succeeding the exit statement. Figure 7 depicts a CFG containing both "exit" and "next" statements.

```
while A='1' loop
  A;                --block of general opeations
  exit when A='0';
  B;                --block of general operations
  next when  X='1'
  C;                --block of general operations
end loop;
D;                  --block of general operations
```

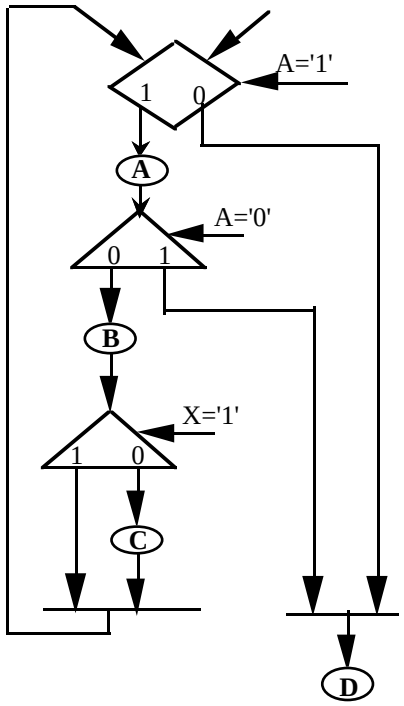


Figure 7: CFG containing "exit" and "next" statements

"Case" statements are mapped directly onto multi-value branch nodes. These nodes have as many outputs as there are VIF case alternatives. Thus, if a case alternative contains a slice of values, one output is generated. Each alternative, with the exception of the "others" alternative, generates a guarded successor node which points to a CFG node, corresponding to the first statement in one of the case alternative branches, and the value required in order to execute this node. If the alternative has a slice of possible values, the guarded value is the higher slice value. Condition guards are always static values. The guarded successor corresponding to the "others" alternative of a "case" statement has no guarded value. This node is taken only if none of the other guards were matched by the result of the "case" condition expression.

Simple "if" statements containing at most two alternatives (if…else…end if;) are mapped onto boolean branch nodes. If more than two alternatives are available (if…elsif…elsif…else…end if;), a set of boolean branch nodes is created. The number of boolean branch nodes created is one less than the number of alternatives if the "else" clause appears, otherwise it is equal to the number of alternatives.

The difference between a "case" statement and an "if…elsif…else…end if;" statement is that the former implies no priority whereas the latter does. This is, in fact, why an "if…elsif" statement is represented by a succession of boolean branch nodes rather than a single multi-value branch. Figure 8 shows the difference more clearly. The alternatives beside each output of the multiple branch node represent guarded successor nodes, the value "E" corresponds to all other possible values of X other than 0 and 1.

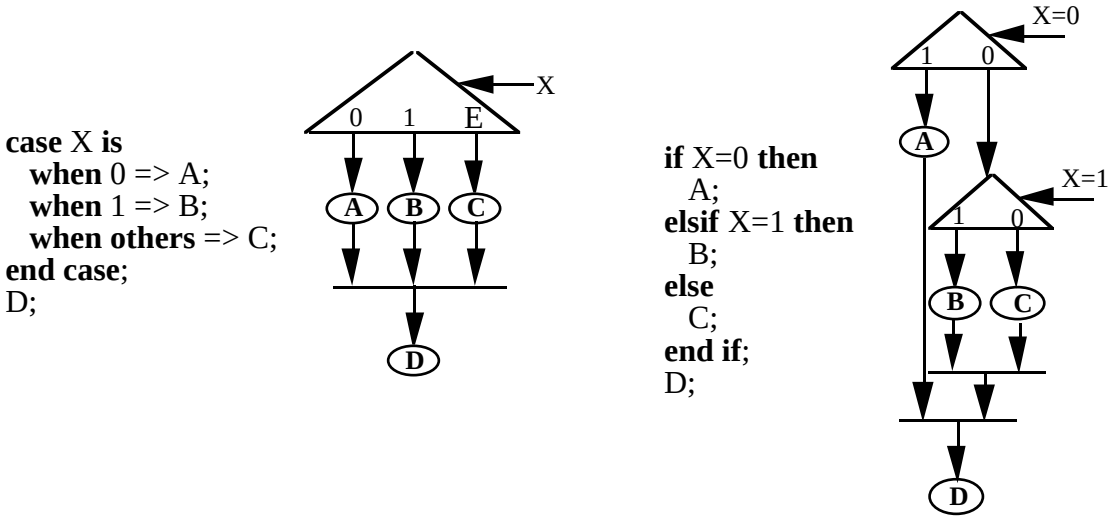


Figure 8: Difference between "case" statement and "if…elsif…else" statement

3.3.2.2 Loop Statements

Loop statements include VHDL "while", "for" and "loop" statements. In our graph domain there is only one node for treating loops (if we ignore for the moment implicit loops due to synchronisation statements and process statements). All "for" loops have either been unrolled or transformed into "while" loops.

Statement 3 of figure 3 is expanded in figure 9. This statement generates a simple DFG that compares the values contained by two variable objects and returns a boolean result to the CFG via an interface node.

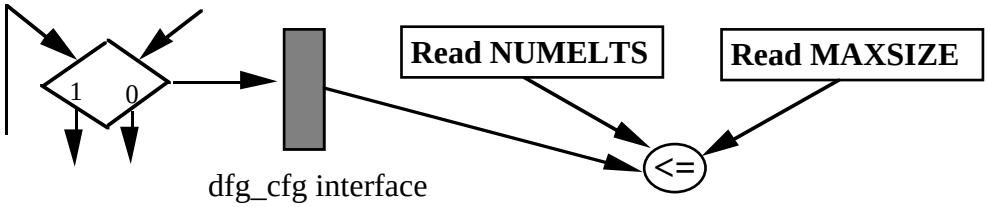


Figure 9: Expanded graphs for statement 3 of figure 4

One point worth noting however, is the fact that the loop boundaries are dynamically evaluated. This kind of loop is generally not accepted by logic synthesis tools and, in order for it to be accepted by higher-level tools, it must contain at least one synchronisation statement in every possible control flow path through the loop.

3.3.2.3 Procedure Call Statements

A procedure call statement is mapped onto a dedicated CFG procedure call node in the graph domain. This node has one successor which points to the first node of a CFG modelling the body of the procedure and a second successor which points to the node representing the statement to be executed immediately after control has returned from the procedure body. An example is shown in figure 10.


```
procedure P (A:in bit,B: out bit) is
begin
  if A='0' then
    B;
  else
    C;
end P;
…
process
begin
  P(X,Y);
  wait;
end process;
```

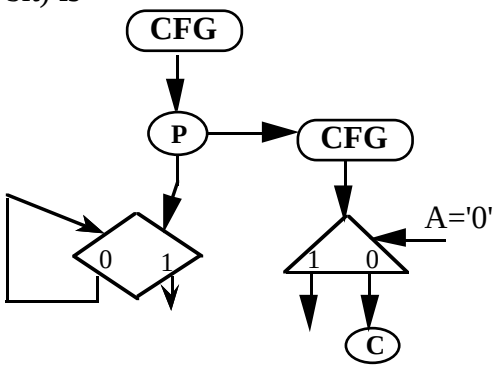


Figure 10: Example of a procedure call node

Synthesis tools usually limit the complexity of procedure bodies. They are normally used to represent frequently executed code or to model either a previously-synthesized functional unit, or a resolution function having no direct hardware equivalent. In the former case, the synthesis tool can replace the call by the procedure body. In the latter case, it is to be treated as a black box and its contents are not relevant to the current synthesis session.

A third case not normally treated by synthesis tools is the use of procedures as a partitioning mechanism. In other words, the body of a procedure is to be synthesised separately from the calling environment. No limitations are made on the VHDL accepted.

Our graph domain caters for all three possibilities. For in-line expansion, the two successors of the procedure call node permit the two corresponding CFGs to be merged. At the same time, the fact that the body and calling environment are separate allows the contents of the body to be ignored. Finally, the generation of an independent CFG (and associated DFGs) for the procedure body facilitate an independent treatment by a synthesis tool. This tool must, of course, add any communication protocols necessary to interface the two partitions.

3.3.3 Data Flow Statements

From the control flow point of view, other VHDL statements create a single CFG general operation node having at most, one successor. The complexity of these nodes is in their data flow. In VHDL, there are two main areas of data flow: the calculation of expressions for assignments and the calculation of expressions for address evaluation. In this section we concentrate on the former.

3.3.3.1 Simple Assignment Statements

In VHDL, an assignment consists of a target and a source. The target can be one or more declared objects or parts of declared objects. The source can be one or more expressions.

In the simplest case, VHDL permits direct assignments from one general used object to another or sources containing simple operator calls whose operands are general used objects or static values. Examples from figure 3 include statements 2, 8, 9, 11, 13 and 21. Figure 11(a) shows the expanded CFG nodes for the first two of these statements. Figure 11(b) shows the representation of a more complex expression involving two simple operator calls. Note that although the operand "B" is used twice, there is only one instantiation of the corresponding operand read node in the DFG. This is because the DFG is constructed as a directed acyclic graph and common sub-expressions are eliminated during generation.

From the synthesis point of view, the simple operator call nodes will map onto functional units, the operand access nodes will map onto registers and the edges will map onto bus segments or multiplexors, depending on the target architecture. Expressions involving more than one simple operator call are usually split into intermediate expressions by the synthesis tool. Therefore, the number of registers required may be increased by the number of intermediate results have to be stored.

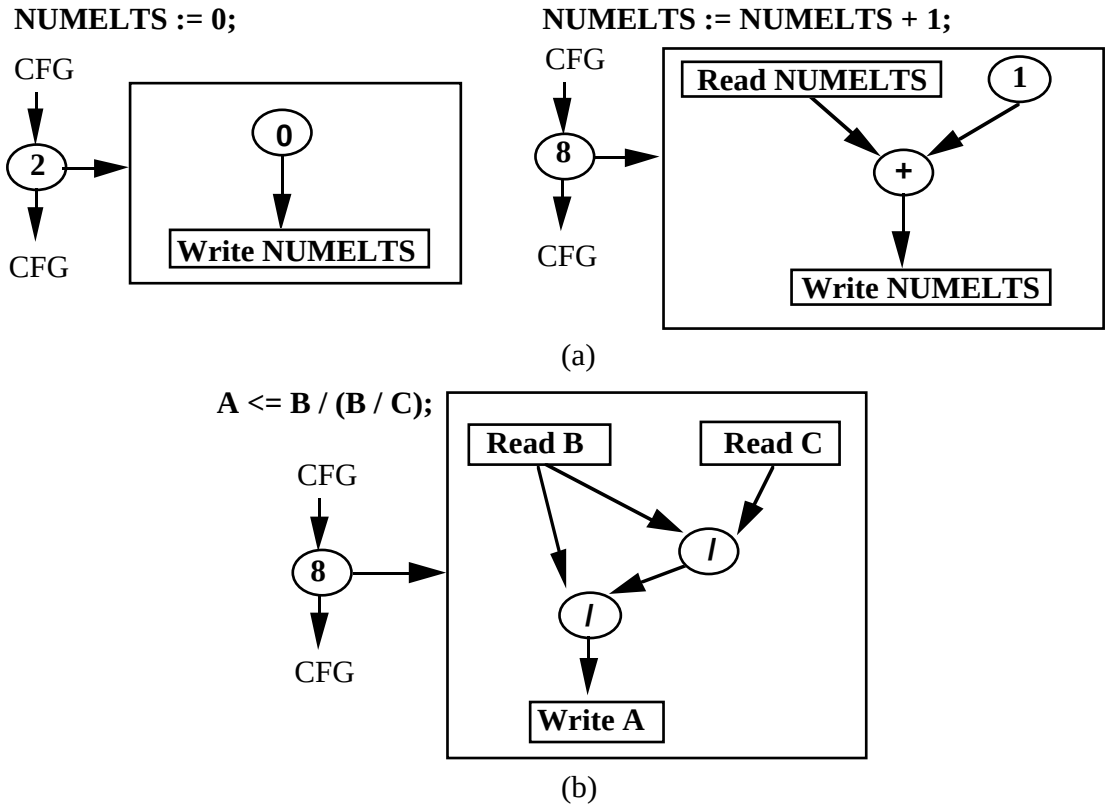


Figure 11: Simple assignments statements (a) Statements 2 and 8 of figure 3 (b) Statement involving two simple operator calls

3.3.3.2 Delayed Signal Assignments, Type Conversions and Qualified Expressions

VHDL also allows the concept of delayed signal assignments through the "after" clause. The time of delay may itself be the result of a complex expression evaluation. In the DFG, this delay information is appended to the final edge before the operand write node. In figure 12, the DFG for statement 6 of figure 3 is shown. The 5ns delay is represented as a constant node pointed to by the final edge.

Other information attached to the final edge before an operand write includes type conversion and qualification information. This is also shown in figure 12 for a qualified expression. The information pertaining to the qualification was generated during compilation and is readily available in the VIF model. As this information need only be read when appropriate, the graph points directly to it through a DFG/VIF interface attached as a property of the edge.

Synthesis tools generally ignore this additional information as they are not easily interpreted in hardware terms. For example, does the time evaluated in an "after" clause represent the minimum, maximum or typical value of the real delay? It is therefore important to separate this information from the main DFG while at the same time we must not penalise synthesis tools that do interpret this information.

ACKOUT <= BIT('1'), '0' **after** 5 ns;

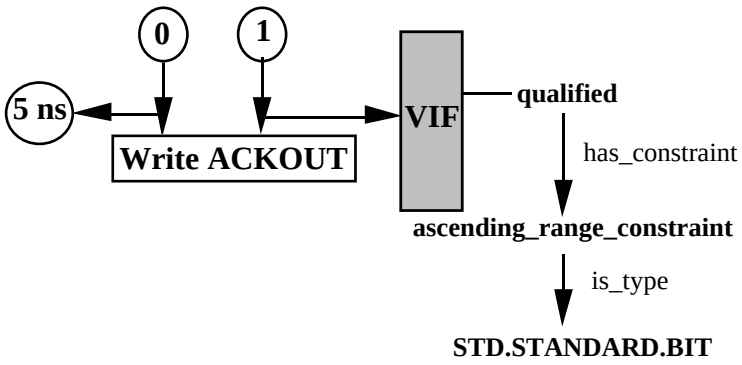


Figure 12: Appending information to assignment edge

3.3.3.3 Function Calls

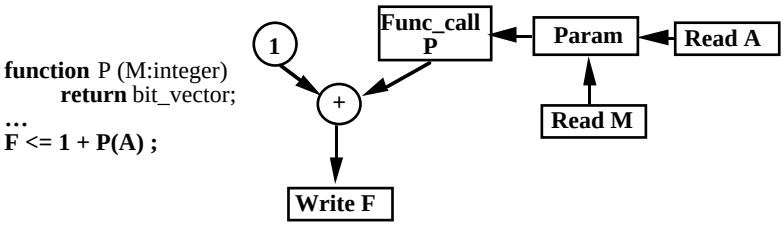
An operand in VHDL may also be a function call. The body of a function may or may not be visible from the calling environment. In addition, the function call, like the procedure call, may be used to instantiate a synthesised functional unit in the DFG and in this case it must be viewed as a black box. In the pre-synthesis graph domain therefore, we must simply evaluate the parameters.

VHDL allows full expressions to be used in the evaluation of actual parameters or parts of actual parameters. It is not even necessary to use all parameters. In addition, VHDL'93 allows type conversion functions to be executed on formal parameters. For each parameter therefore, the DFG contains a parameter node which points to the evaluation of both the actual and the formal.

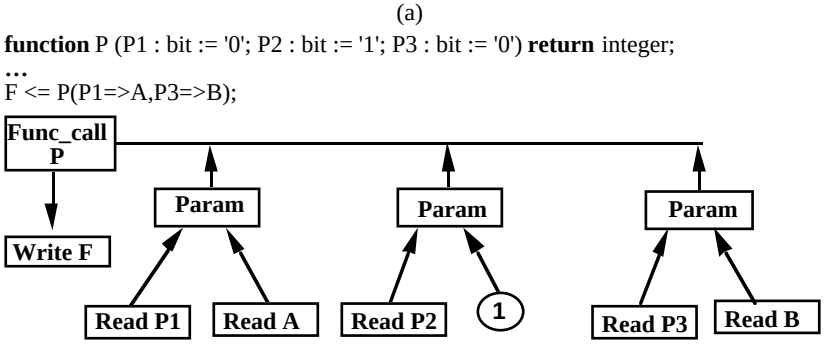
Figure 13(a) shows the declaration and call of a function containing one parameter. Note that the parameter node evaluates both actual and formal expressions. If the formal parameter was type converted, this would appear as a property of the edge pointing from the parameter node to the formal parameter expression.

In figures 13(b) and 13(c) the function is called without the second parameter. In one case, it is completely omitted, in the other the "open" key word is used. Note that the DFG for both cases is identical with the default value being used as the actual parameter.

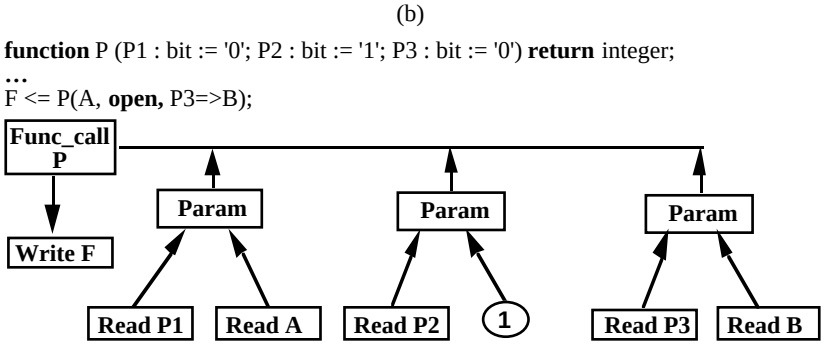
Figure 13(d) shows another possibility permitted in VHDL, that of partially associating an unconstrained formal parameter. The partial associations are gathered together through the DFG merge read node. For each partial association, a segment of the merge read node points to an expression evaluating the partial actual and to a DFG address node that in turn points to an expression evaluating the partial address of the formal with which the actual is associated. Address nodes will be discussed in more detail in the following section.



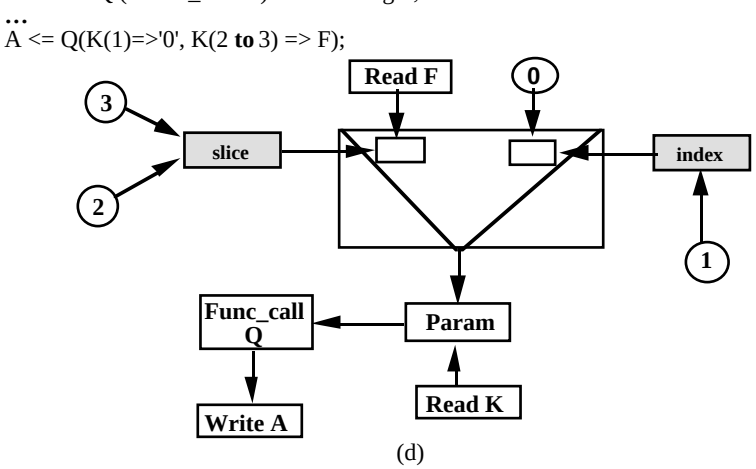
(a)



(b)



(c)



(d)

Figure 13: Using function calls as operands (a) General case (b) Second parameter omitted (c) Second parameter unconnected (d) Parameter partially associated

3.3.4 Address Evaluation

In the previous section, all operands were either function calls or scalar object types. Sometimes, as illustrated by figure 3, it is more convenient to group blocks of data into the same structure. VHDL has some very powerful data typing facilities to implement this. Unfortunately, synthesis tools do not take advantage of this facility, mainly due to the lack of an efficient synthesis-oriented representation.

All non-scalar VHDL operands that are partially addressed are mapped onto array read and write nodes in the graph domain. This is valid for operands declared as arrays (1 or multi-dimensional), enumerated types or even records (as a record can be seen as an array with elements of varying length). The difference between these operands lies in the manner in which the array index (the address) is calculated. VHDL also permits multiple and array aggregate assignments, all of which must be considered.

The graph domain contains six DFG nodes in order to facilitate the address evaluation. In the case of a dynamic address evaluation, the DFG points to the VIF representation of the expression which contains all of the available information.

3.3.4.1 Index Addressing

The most simple form of addressing is the simple indexing of array objects, as in statement 16 of figure 3. In this case, a single index address node will be used for both accesses of array "RAM". In both cases, the index address node points to the expression evaluating the index, as depicted in figure 14(a). For multi-dimensional arrays, as many index address nodes as there are dimensions will be used. These nodes point to a list of expressions, each calculating the address corresponding to a single dimension. This is shown in figure 14(b). Note that the same representation is used for both type of multi-dimensional array access.

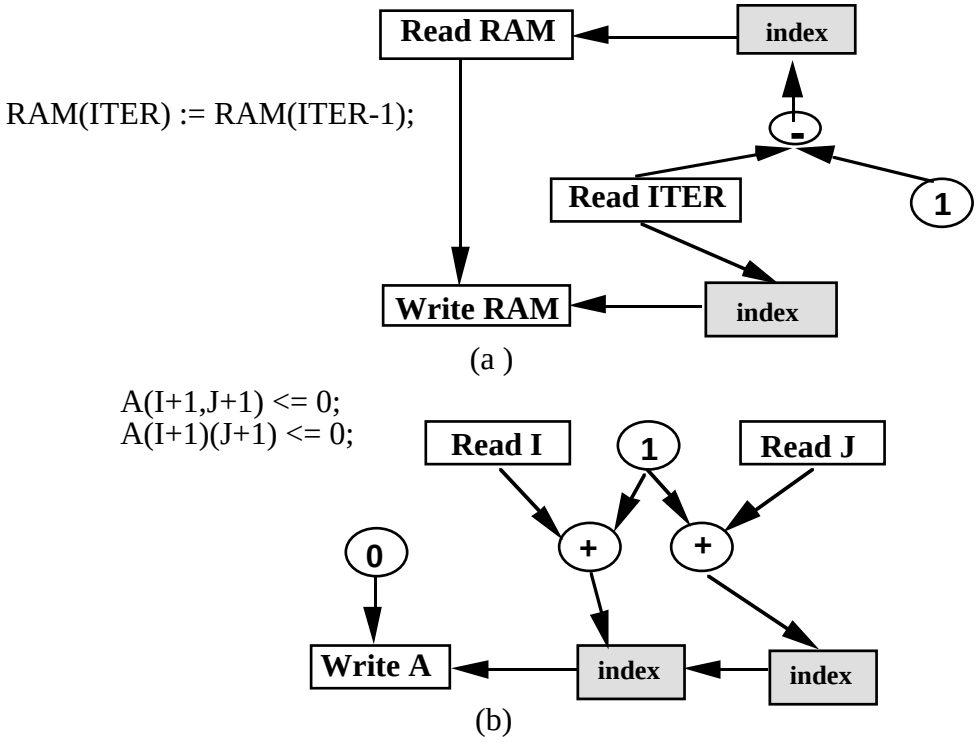
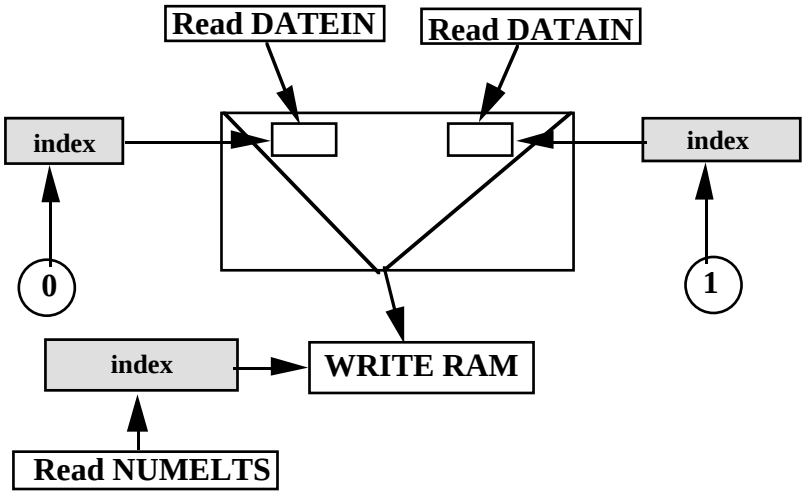


Figure 14: Indexing array objects (a) 1-dimensional arrays (b) N-dimensional arrays

3.3.4.2 Record Addressing

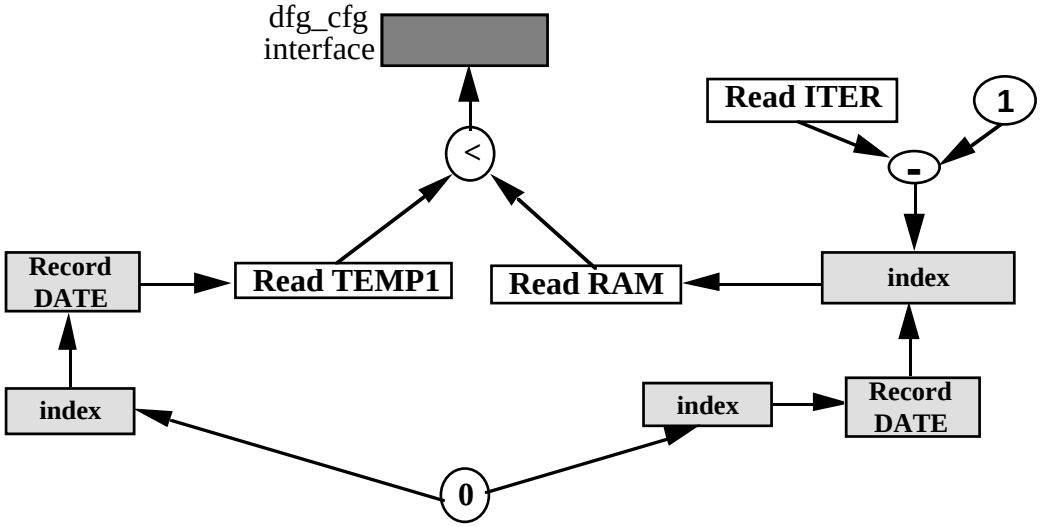
In figure 3, the array "RAM" contains a set of record items, each having two fields. These fields can be accessed individually or simultaneously. Consider statement 5 of figure 3, whose DFG is shown in figure 15(a). In this statement, we simultaneously assign both fields of the record situated in address "NUMELTS" of array RAM. In the graph domain, all simultaneous reads and writes are modelled by the DFG merge read and write nodes. In hardware, this node represents a bus merge or split function. The merge read node of figure 15(a) has two segments, one for each field. Each segment has a pointer to an expression representing the value to be read. In this case, it is a simple operand read. Each segment also has a pointer to an address node that evaluates the partial address of the operand to be written. In the case of records, these will always be constant values corresponding to the position of the field in the record declaration.

RAM(NUMELTS) := (DATEIN, DATAIN);



(a)

if TEMP1.DATE(0) < RAM(ITER-1).DATE(0) **then**…



(b)

Figure 15: Addressing record objects (a) Simultaneous access of all fields (b) Accessing one field

In figure 15(b), we see the DFG corresponding to the condition evaluation of statement 15 of figure 3. On the left hand side of this statement, we access an element of an array that is stored in one of the record fields. Thus, two addressing nodes will be required, one to calculate the record address (the field) and one to calculate the field's array address. We can interpret the nodes as accessing index "0" of the array stored in record address "DATE" of record "TEMP". The right hand side of the expression is slightly more complicated as the record itself is an element of an array. Thus three address nodes will be necessary. We interpret the nodes shown for this operand as accessing index "0" of record address "DATE" situated at address "ITER-1" of array "RAM".

Another form of record addressing is to use allocated pointers. This is discussed in the next section.

3.3.4.3 Allocated and Access Addressing

Another form of addressing in VHDL permits the use of accesses. Consider the code shown in figure 16.

```
type R1 is record
   F1 : integer;
end record;
type P is access R1;
type T is access integer;
…
process
   variable P2,P3 : P;
   variable T2 : T;
begin
   T2.all := 1;              --1
   P2.F1 := 2;               --2  same as P2.all.F3.F2(0 to 2)…
   P3.all := P2.all;         --3
   wait;                     --4
end process;
…
```

Figure 16: VHDL showing access and allocated addressing

The first three statements of figure 16 use access types to refer to different memory elements. An access declaration may be seen as a pointer to an address. Thus, it is not a memory location corresponding to the declaration that is to be modified but rather the memory location pointed to by the declaration. This is why, when modelling access types the array read and write nodes do not have a name corresponding to a VHDL item. Figure 17 shows the DFGs generated for the first three statements of the process in the above code segment.

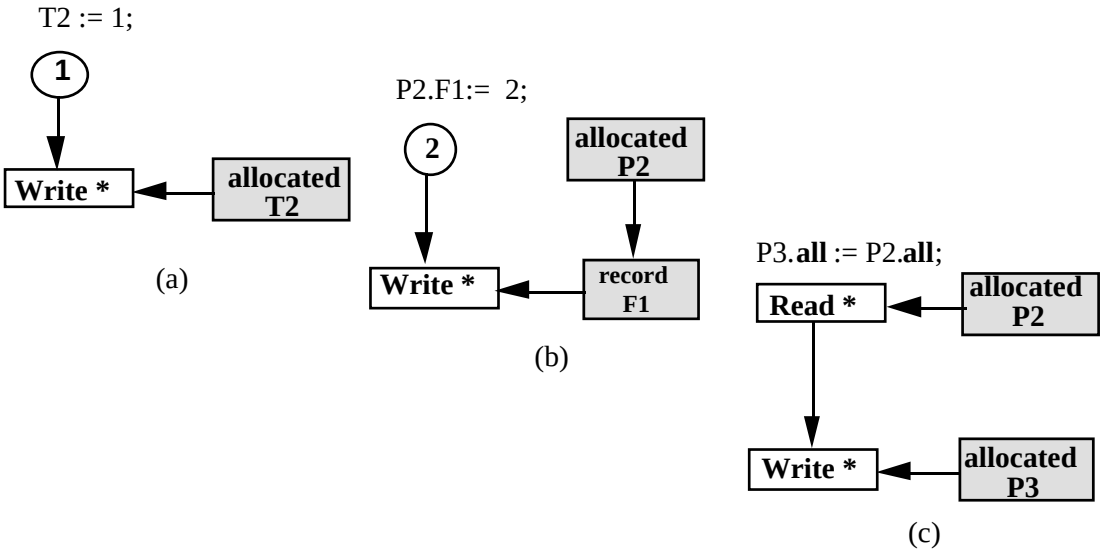


Figure 17: Modelling allocated and access addressing (a) Pointer to a memory element storing an integer value (b) Pointer to record field (c) Allocated memory accesses

3.3.4.4 Slice Addressing

VHDL allows slices of arrays to be simultaneously accessed, as in statement 19 of figure 3. Figure 18 shows the DFG model corresponding to statement 19. The DFG merge read node is used to combine the different values to be assigned to the RAM address. This type of aggregate addressing is discussed in more detail later. A DFG slice node is used to point to the expressions that evaluate the upper and lower bounds of the array slice. If the bounds cannot be evaluated during graph generation, an interface to the VIF is created which points to all available information at compile and/or elaboration time.

RAM(ITER).DATE(1 **to** 2) := (2=>99,1=>13);

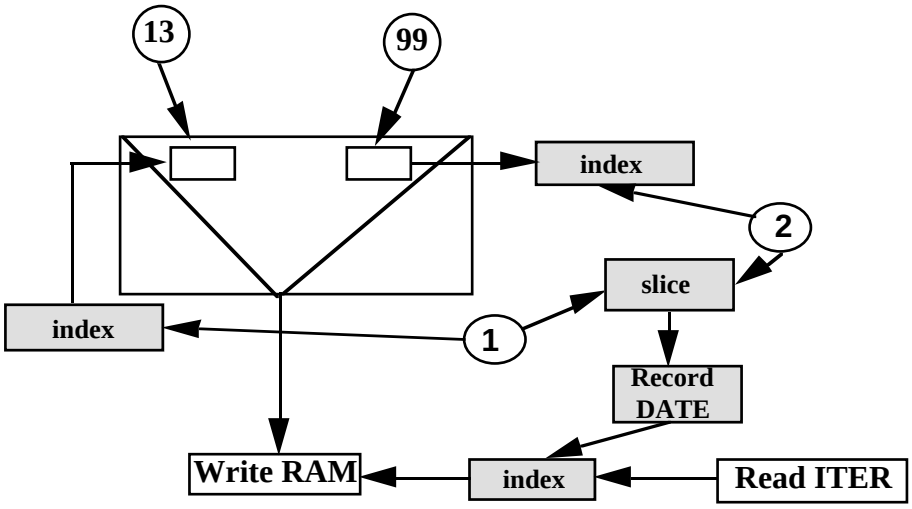


Figure 18: Writing to an array slice: statement 19 of figure 3

Figure 19 shows three different array slice accesses. In figure 19(a) and figure 19(b) the bounds of the slice can be evaluated. In figure 19(c) however, we assume that the attribute A'RANGE cannot be evaluated statically. In this case the slice address attribute "static_bounds" is set to "false" and no DFGs are created for the bound attributes. However, the slice address node also points to the VIF representation of the slice expression. The information necessary to evaluate the range attribute at execution time is available in the VIF representation.

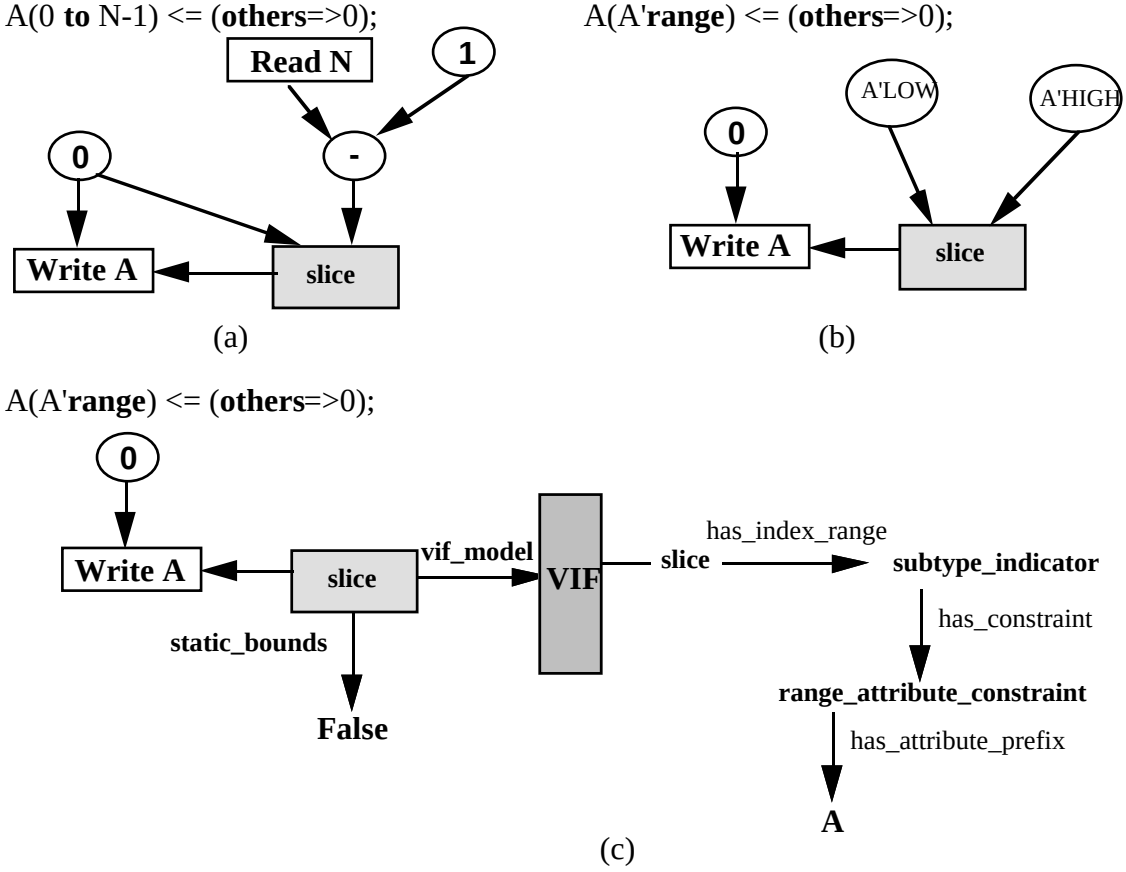


Figure 19: Slice addressing of arrays (a) simple slice (b) static range attribute (c) dynamic range attribute

3.3.4.5 Aggregate Addressing

As shown in figure 18, VHDL allows individual elements of a non-scalar object to be simultaneously accessed. This is known as aggregate addressing and is not unlike the partial association of sub-program parameters discussed in previously.

Figure 20 gives some examples of aggregate addressing. In figure 20(a), The elements of the slice are assigned a value according to their position. In other words, array element A(0) is assigned 4, A(1) 5 and A(2) 6. This is in contrast to figure 18 where each element of the aggregate was assigned by its name. The result is the same and thus the graph domain treats position aggregates and named aggregates in the same manner. In order to model aggregate reads we use a merge read node that concatenates different expressions and assigns them to the target.

Figure 20(b) shows how the "others" clause is interpreted in an aggregate expression. This is in fact the DFG corresponding to statement 20 of figure 3. In this case, "DATA(0)" receives the value '0' and all elements of the slice "DATA(1 to 255)" receive the value '1'.

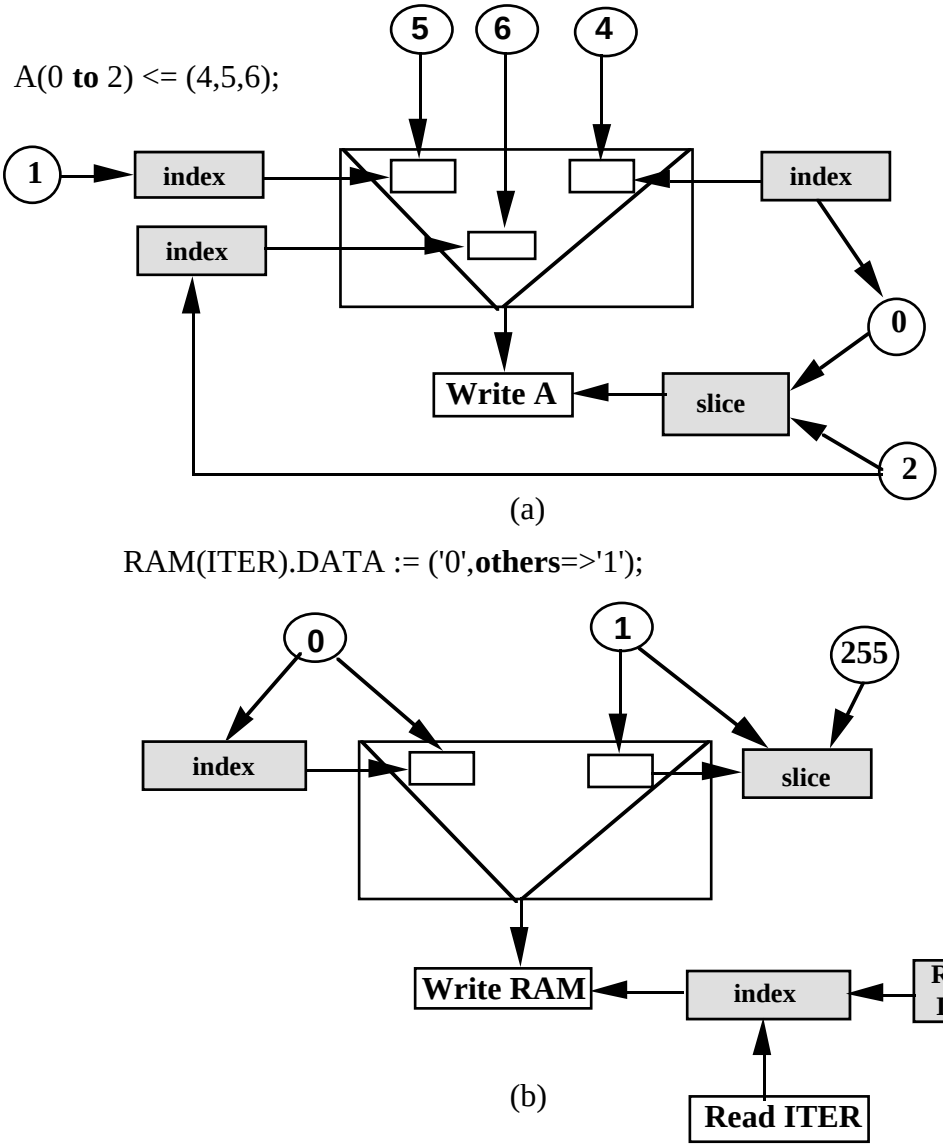


Figure 20: Using the merge read node to resolve positional array aggregate assignments

In the statement of figure 21, the target is an aggregate of four different signals. In other words, four different memory writes are being executed simultaneously. This is modelled by the merge write node, each signal being assigned a segment of the merge node. On the right hand side, we note that elements in position 0 and 3 and elements in position 1 and 2 of the target aggregate receive the same values (6 and 5 respectively). This is modelled by the merge read node of figure 21. Note that one segment of the merge read node has a slice address corresponding to the slice "1 to 2". The others (0 and 3) have been given individual index addresses.


```
type BIT_4 is bit_vector(0 to 3);
…
(S1,S2,S3,S4)<= BIT_4'(1 to 2 => '0',  0 | 3 => '1');
```

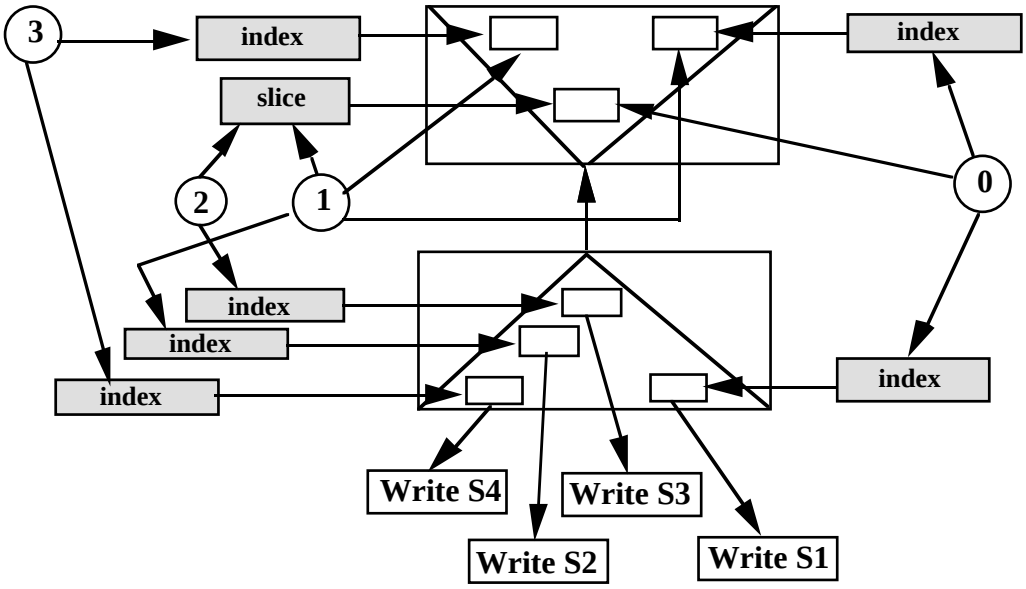


Figure 21: Using merge read/write nodes to model array aggregates

3.3.5 Optimising CFG Construction

After a VHDL unit has been compiled by LVS, all locally static expressions are evaluated. This means that, rather than constructing a graph representing the input expression, we can construct a graph for the value of this expression. In terms of data flow this means replacing whole DFGs by a simple constant node. In terms of control flow, if we can identify locally static condition expressions, it means the possibility of eliminating entire execution paths. This optimisation can easily be extended to globally static expressions when the elaboration phase is complete.

Take for example the following code:

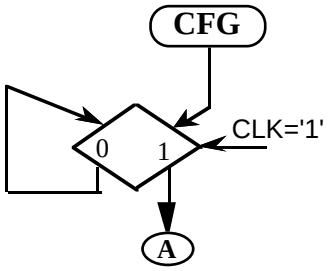
```
if GENPARAM=1 then
   X;      --execution path containing N1 statements
else
   Y;      --execution path containing N2 statements
end if;
```

Suppose we know that GENPARAM=1. Then we need not generate CFG nodes for the N2 statements of the "else" branch. Subsequently, we do not need to generate a boolean branch node as the same branch is always taken.
Similar tests are performed on all VHDL control flow statements ("case", "while", "loop", "exit",…).

3.3.5.1 Branch statements

Apart from "if" statements, "else" and "next" statements also generate boolean branch nodes. These statements can be treated similarly to "if" statements, as shown in figure 22. If the condition of an "exit" statement resolves to true or there is no condition then no node is generated but the successor of the node preceding the "exit" statement is modified so that it is the first node outside the containing "loop" statement. Similarly, a "next" statement having a condition that evaluates to true does not generate a node but rather modifies the successor of the preceding node so that it points to the containing "loop" node. If the condition is false, the preceding node simply points to the node succeeding the "next" statement.

```
process
begin
  wait until CLK='1';
  if TRUE then
    A;
  else
    B;
  end if;
end process;
```




```
process
begin
  wait until CLK='1';
  while X=Y loop
    A;
    next;
    B;
  end loop;
  C;
end process;
```

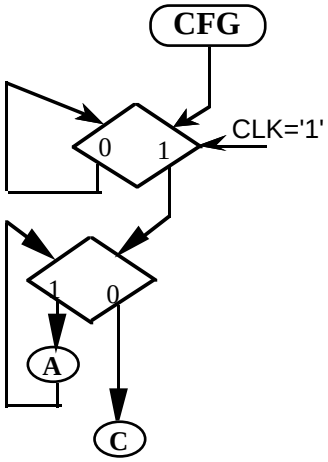


Figure 22: CFGs generated for branch statements with static conditions

3.3.5.2 Case statements

If a case condition is static then only one of the branches will be taken. This is reflected in the graph created for such a case statement as shown in figure 23. Although this code is not very realistic, we can imagine the situation where X is a generic parameter evaluated during elaboration.


```
process
begin
  wait until CLK='1';
  X := 1;
  case X is
    when 0 => A;
    when 1 to 5 => B;
    when 6 => C;
    when others => D;
  end case;
  E;
end process;
```

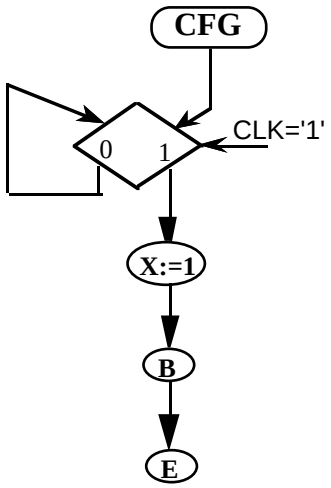


Figure 23: CFG generated for case statement with static condition

3.3.5.3 Loop statements

If a loop condition can be evaluated we can not only remove the CFG loop node, but also some of the statements within the loop as shown in figure 24. It should be noted that the "loop…end loop" statement never generates a node. Flow of control edges within the loop are modified. Another situation that often arises with loops is that the number of iterations is known. In this case the loop can be unrolled. However, this is not always a desirable effect therefore we do not unroll loops at this stage but leave it as an optimisation option for the user.

```
process
begin
  wait until CLK='1';
  while TRUE loop
    A;
    exit;
    B;
  end loop;
  C;
end process;
```

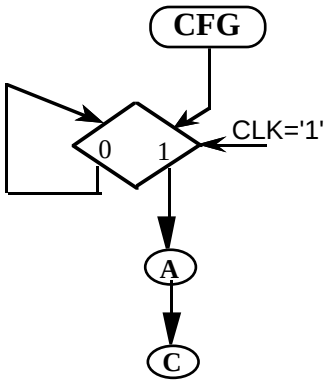


Figure 24: CFG generated for loop statement with static condition

3.4 Graph Partitioning

Once the graphs have been generated, we can perform some pre-synthesis tasks normally executed by synthesis tools. One of these tasks is the partitioning of the control flow graph into sets of contiguous nodes. The most popular partitions are basic blocks and execution paths. From a synthesis point of view, if there are no data dependencies or hardware limitations, each partition can execute all of its nodes in parallel. From a software point of view, partitioning into independent blocks enables a large class of high-level transformation algorithms to be performed, improving the overall execution time and resource requirements.

3.4.1 Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. To generate basic blocks for full VHDL, the basic algorithm presented in "*Compilers: Principles Techniques and Tools*" by Aho et al was extended slightly.

The first step is to get all block leaders. A block leader is the first node in a basic block. For our CFG these are:

- The first node.
- All cfg_loop nodes.
- Any node that is the successor of a cfg_loop, cfg_boolean_branch, cfg_multiple_branch or cfg_wait node.

For each leader, its basic block consists of all nodes upto but not including the next leader or a node with no successors. Figure 25 shows the basic block representation of the CFG of figure 4. The numbers in each block correspond to the number associated with each node in figure 4.

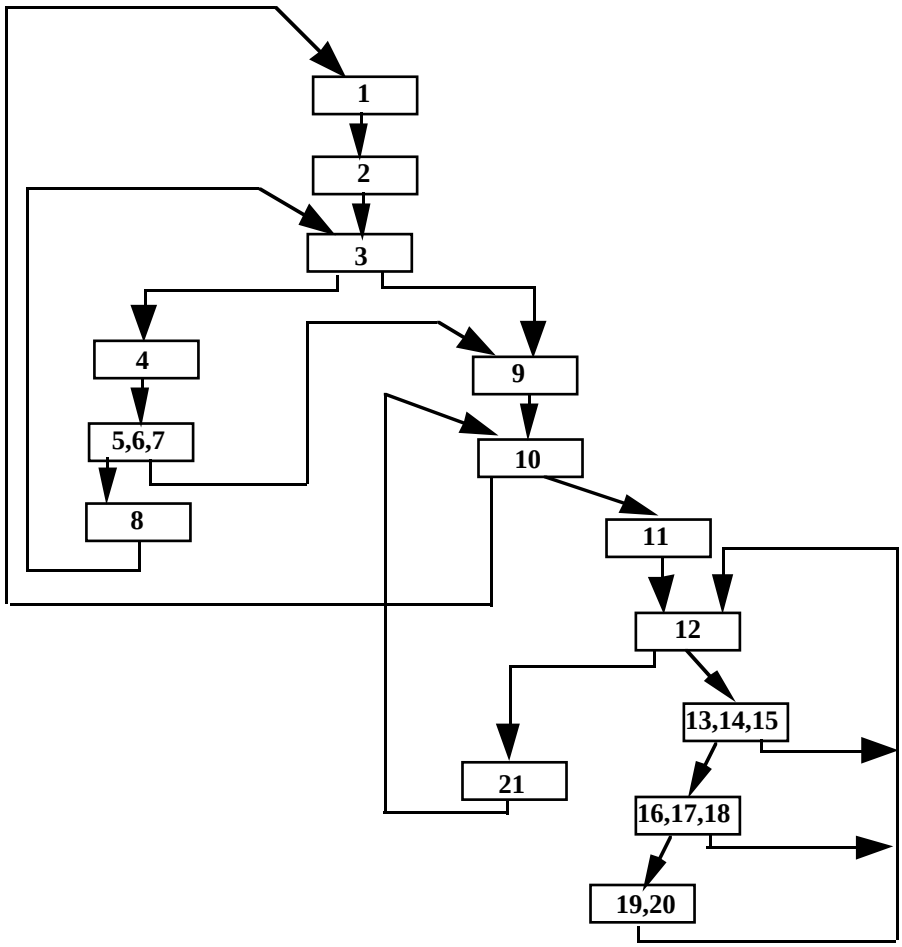


Figure 25: Basic block representation of VHDL CFG of figure 4.

For each basic block, a new DFG is created by merging the individual DFGs of the individual CFG nodes that constitute the block. This new DFG is constructed as a directed acyclic graph which automatically enables us to eliminate common sub-expressions between the nodes of the CFG. For example, suppose we have the nodes representing the following code in a basic block:

```
X := B+C;
Y := B+C;
```

The DFG constructed for this basic block is shown in figure 26. Note that only one add operation node is necessary and the result is written to both X and Y.

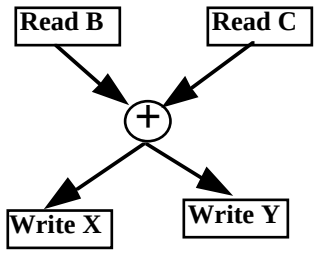


Figure 26: Optimising resource sharing by eliminating common sub-expressions

3.4.2 Execution Paths

Execution paths are another way of partitioning CFGs both for optimisation and synthesis purposes. As their name suggests, execution paths gather together CFG nodes that can be executed during one complete pass of the graph. Thus, at any node with more than one successor, a new set of paths is generated, one for each successor. Nodes can therefore appear in more than one path. DFGs for each path are constructed as for basic blocks. The execution paths for the CFG of figure 4 are shown in figure 27. Each node of the CFG is represented by a circle containing the corresponding number. The box at the end of each path contains the number of the node that will execute immediately after the execution of the current path. It is known as the path successor. All paths start with the first node. Successor nodes are sequentially appended until either a node has no successor or the next node already appears in the path.

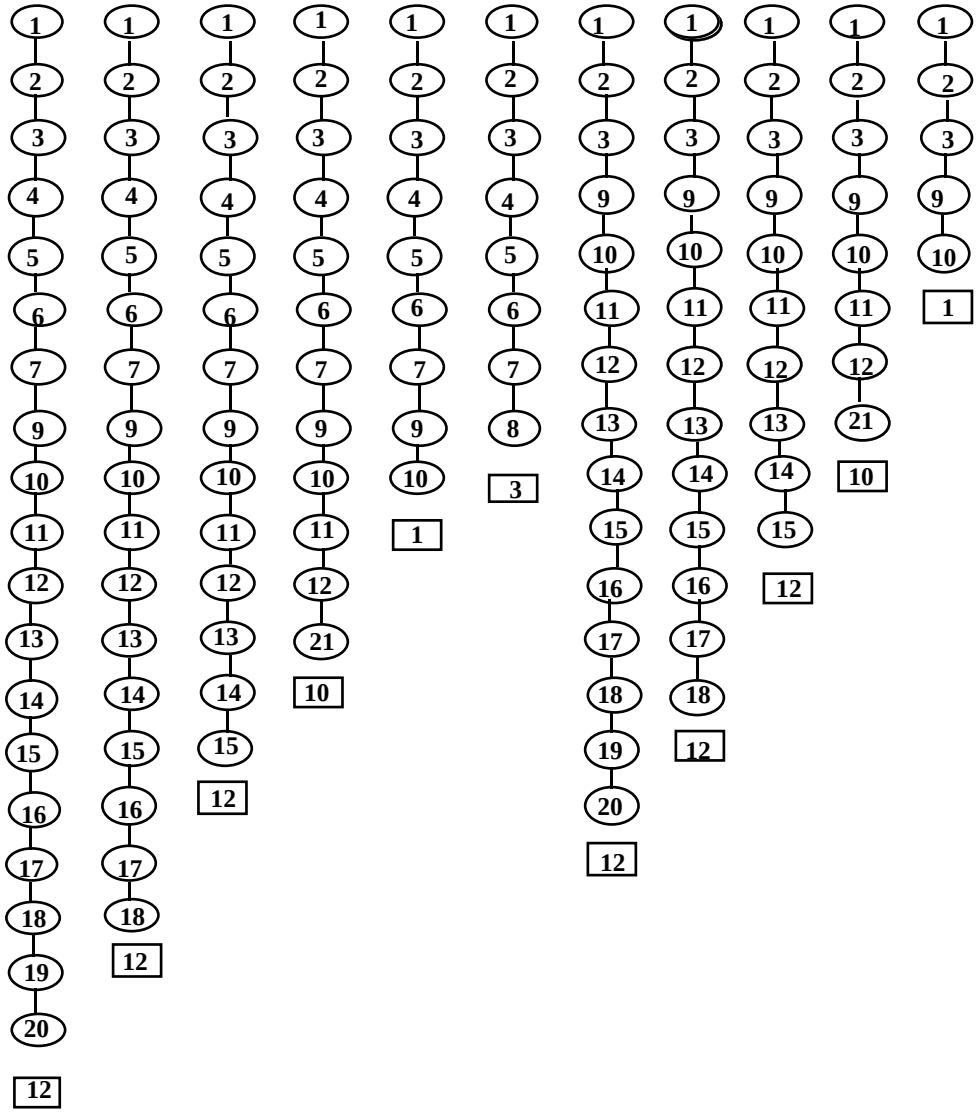


Figure 27: Execution paths for CFG of figure 4

The next step performed by **GRAPHGEN** is to eliminate duplicated paths. All path successors will in fact become leaders of a new set of paths. We can therefore remove all sub-paths beginning with this node from the original paths (except, of course, if this node is already the path leader). The final set of paths can be see in figure 28.

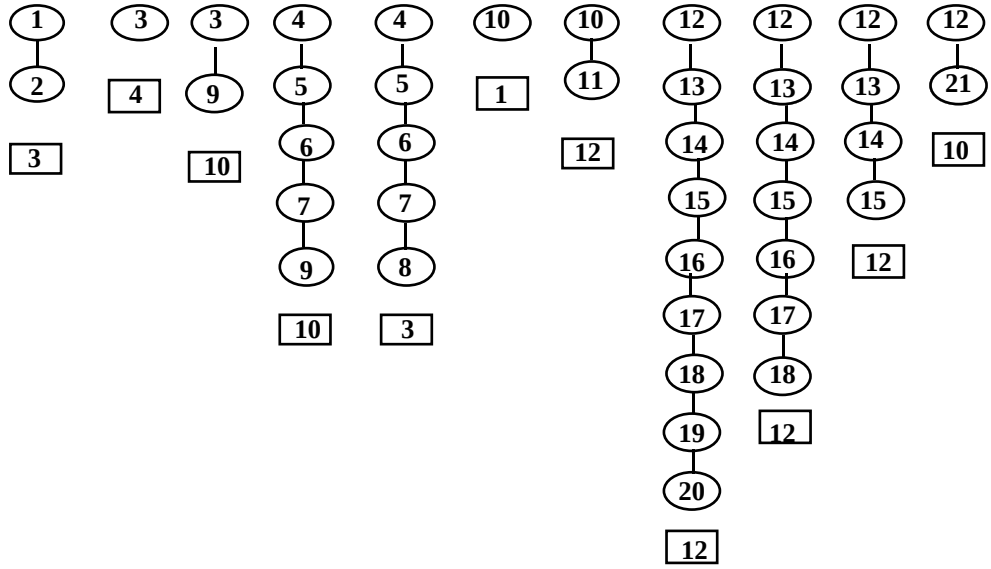


Figure 28: Final set of paths for CFG of figure 4

Synthesis tools can now start scheduling these nodes. The first property we can identify is that there will be at least 5 states in the controlling FSM, one for each path leader.

4. PROCEDURAL INTERFACE

The procedural interface to **GRAPHGEN** can be split into two parts: **GRAPHGEN**'s procedural interface and a more global procedural interface that contains more general subprograms.

4.1 GRAPHGEN Procedural Interface

To use **GRAPHGEN**'s procedural interface, the application must be linked with **graphgen.a, lvskernel.a** and the extended **uvif.a** as shown in section 2. The **Makefile** in the **Example** directory of the **GRAPHGEN** package shows how this can be done. The script **gphtest** shows how to build the extended schema.

Note: The order on linking the archives is very important.

The application will consist of one or more C files that must be compiled. To have access to the functions and procedures of **GRAPHGEN**'s procedural interface, the application must include the directive:

```
#include "graphgen.h"
```

where **graphgen.h** is the header file in the **Release** directory of the **GRAPHGEN** package.

4.1.1 Primitive Types

GRAPHGEN introduces one new primitive type, gphPathOptions.

typedef **gphPathOptions** {
NoPaths,
SimplePaths,
OptimizedPaths
ControlFlowPaths
}

This type is used for specifying the type of paths to generate. It is used as one of the parameters to *gphGraphGen()*. Each element can be specified as a parameter on the command line of the executable.

NoPaths:	No paths generated, path generation options do not appear on command line.
SimplePaths:	Classic execution paths for synthesis, **-ps** option on command line
OptimizedPaths:	Simple paths cut to create new paths at wait statements and to identify and isolate all path headers, **-p** or **-po** option on command line.
ControlFlowPaths:	Different possible flows of control between the 1st node and the last node of a set of sequential statements, **-pc** option on command line.

4.1.2 Procedure gphGraphGen

Parameters

Name	Mode	Type
LibNode	in	vifLibraryUnit
gphBasicBlocks	in	vifBoolean
gphExecutionPaths	in	vifBoolean
PathOptions	in	gphPathOptions
GenerateDFG	in	vifBoolean
ControlInDFG	in	vifBoolean
CreateEquivalentProcess	in	vifBoolean

Description

Generate a control flow graph (CFG) structure for different concurrent statements in the design unit pointed to by **LibNode**. If **gphBasicBlocks** is True, basic blocks are extracted for each CFG. If **gphExecutionPaths** is True, execution paths are extracted for each CFG. The type of execution path is determined by the value of **PathType:**

```
switch (PathType){
        when NoPaths            : break
        when SimplePaths        : simple paths
        when OptimisedPaths     : optimised paths
        when ControlFlowPaths   : control flow paths
}
```

For all CFG structures (nodes, basic blocks, paths), if **gphGenerateDFG** is True, a corresponding data flow graph (DFG) is generated. If **ControlInDFG** is True all data flow is executed in the DFG (equivalent to the **-d** option of the executable - see §2.3). If **ControlInDFG** is False, all equality and inequality expressions having one locally static operand will be executed in the CFG (no DFG will be built).

If **CreateEquivalentProcess** is True, concurrent statements selected_signal_assign, conditional_signal_assign, concurrent_assert and concurrent_procedure_call will first be transfromed into their equivalent processes before being treated. If it is False, this is equivalent to setting the **-n** option in the executable. The graphs will be attached to the concurrent statements themselves.

If **GRAPHGEN** has already been executed on the unit, this procedure will return without performing any transformations. To always force the execution of **GRAPHGEN**, use **gphForceGraphGen()**.

4.1.3 Procedure gphForceGraphGen

Parameters

Name	Mode	Type
LibNode	in	vifLibraryUnit
gphBasicBlocks	in	vifBoolean
gphExecutionPaths	in	vifBoolean
PathOptions	in	gphPathOptions
GenerateDFG	in	vifBoolean
ControlInDFG	in	vifBoolean
CreateEquivalentProcess	in	vifBoolean

Description

Generate a control flow graph (CFG) structure for different concurrent statements in the design unit pointed to by **LibNode**. If **gphBasicBlocks** is True, basic blocks are extracted for each CFG. If **gphExecutionPaths** is True, execution paths are extracted for each CFG. The type of execution path is determined by the value of **PathType:**

```
switch (PathType){
        when NoPaths            : break
        when SimplePaths        : simple paths
        when OptimisedPaths     : optimised paths
        when ControlFlowPaths   : control flow paths
}
```

For all CFG structures (nodes, basic blocks, paths), if **gphGenerateDFG** is True, a corresponding data flow graph (DFG) is generated. If **ControlInDFG** is True all data flow is executed in the DFG (equivalent to the **-d** option of the executable - see §2.3). If **ControlInDFG** is False, all equality and inequality expressions having one locally static operand will be executed in the CFG (no DFG will be built).

If **CreateEquivalentProcess** is True, concurrent statements selected_signal_assign, conditional_signal_assign, concurrent_assert and concurrent_procedure_call will first be transfromed into their equivalent processes before being treated. If it is False, this is equivalent to setting the **-n** option in the executable. The graphs will be attached to the concurrent statements themselves.

4.2 Global Procedural Interface

Subprograms used by all LVS toolbox products are supplied in the **lvsexpr.h** file located in the **Utils** directory of the distribution package. To use these functions, add the following line to the C code:

```
#include "lvsexpr.h"
```

For more information about other elements of the LVS toolbox, contact sales@leda.fr.

In the global procedural interface, there is a set of functions used to recover information generated by **GEME**, the GEneric Mapper and Elaborator (another element of the LVS toolbox). However, if **GEME** is not present in the current user-application, the **GEME** Schema Extension is, naturally enough, not present. Thus, we have developed part of the global procedural interface that consists of a set of function pairs. Each pair has the same name and profile (same header file), but whereas one works on the **GEME** Schema Extension, the other is used if **GEME** is not included. This set of functions is in **lvsexpr.h**.

Function **elbGetScalarValue**:

For a scalar expression, if it is locally static, its value is returned in 32 bits.

If it is globally static and **GEME** has been executed, a 32 bit value corresponding to the expression is returned.

If it is globally static and **GEME** has not been executed, no value is returned.

If it is not globally static, no value is returned.

Function **elbGetCompositeValue**

For a composite expression, if it is locally static, its value is returned in 32 or 64 bits.

If it is globally static and **GEME** has been executed, its value is returned in a predefined data structure (see **GEME** Schema Extension). The value is split into 32 bit blocks.

If it is not globally static, no value is returned.

For example, look at the following code:

```
entity E is
   generic (G : INTEGER);
   port ( CLK : in BIT;
          CNT : in INTEGER;
          DAT : out INTEGER);
end E;
architecture A of E is
begin
   P:process
      variable V : INTEGER;
   begin
      wait until CLK='1' and CLK'EVENT;          --1
      V := CNT;                                  --2
      if G = 1 then                              --3
         while V > 0 loop                        --4
            DAT := DAT + V;                      --5
            V := V - 1;                          --6
            wait until CLK='0' and CLK'event;    --7
         end loop;                               --8
      else                                       --9
         DAT <= 0;                               --10
      end if;                                    --11
   end process P;
end A;
```

Figure 29: Example showing benefit of using GEME with GRAPHGEN

Assume that we want to execute **GRAPHGEN** to determine the execution paths in the process. **GRAPHGEN** will try and optimise the paths during their construction. For example, if there is a locally static expression, **GRAPHGEN** will replace it by its value, thereby reducing the size of the data-path (represented by the DFG). To verify if an expression is locally static, **GRAPHGEN** executes one of the functions **elbGetScalarValue** or **elbGetCompositeValue**. In our example, this leads to nothing and the execution paths are as follows:

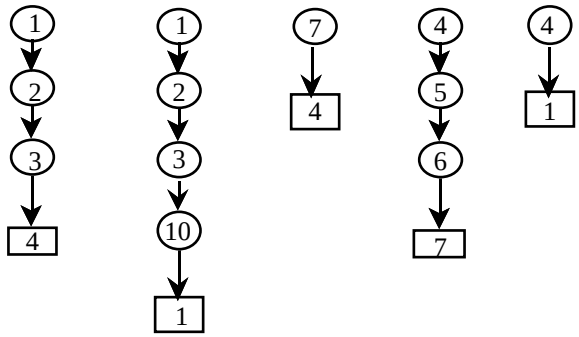


Figure 30: Paths extracted when GEME is not present

Now assume that GEME is included in the user-application and has already been executed. We note that the if expression on line 3 of the code is globally static (*G=1* where *G* is a generic parameter). If we execute **GRAPHGEN** again, it will still execute the functions **elbGetScalarValue** and **elbGetCompositeValue** but this time, their functionality will include retrieving values of globally static expressions from the GEME Schema Extension. In our example, the value of the **if** expression is known when the graphs are being constructed and therfore **GRAPHGEN** can eliminate the branch that is not taken in the above example. This leaves us with the following set of paths for $G \neq 1$.

Figure 31: Paths extracted with GEME present

This difference in functionality is completely transparent to **GRAPHGEN** (and to any user application). **The same functions are called whether GEME is included ot not.**

4.2.1 Function elbGetScalarValue

Parameters

Name	Mode	Type
Expression	in	vifNode
Value	out	vifInteger32*
	<return>	vifBoolean

Description

If the expression represented by **Expression** is scalar, this function tries to evaluate it. If it can be evaluated, its value is assigned to ***Value** and True is returned. If it cannot be evaluated, ***Value**=0 and False is returned.

4.2.2 Function elbGetCompositeValue

Parameters

Name	Mode	Type
Expression	in	vifNode
Value	out	Elaborated_Data*
ValLength	out	vifInteger32*
	<return>	vifBoolean

Description

If the expression represented by **Expression** is composite, this function tries to evaluate it. If it can be evaluated, its value is pointed to by the data structure ***Value** and True is returned. The parameter **ValLength** points to the size of **Value** in 32 bit blocks.

If the expression cannot be evaluated, ***Value**=NULL, ***ValLength**=0 and False is returned.

4.2.3 Function absExprHasValue

Parameters

Name	Mode	Type
Expr	in	vifNode
	<return>	vifBoolean

Description

Function returns True if the expression **Expr** has a value, otherwise it returns False.

If GEME is included, the value returned is True if the expression is globally static. If not, True is returned only if the expression is locally static.

4.2.4 Function absExprAreEqual

Parameters

Name	Mode	Type
N1	in	vifNode
N2	in	vifNode
	<return>	vifBoolean

Description

Returns True if the values of expressions **N1** and **N2** can be evaluated and are equal.

4.2.5 Function absExprIsZero

Parameters

Name	Mode	Type
Expression	in	vifNode
	<return>	vifBoolean

Description

Returns True if the value of expression **Expression** is 0.

4.2.6 Function elbGetValue

Name	Mode	Type
Expression	in	vifNode
	<return>	vifBoolean

Description

Try to evaluate any expression, scalar or composite. The type of the expression is evaluated and, depending on the result, *elbGetScalarValue()* or *elbGetCompositeValue()* is called. The result is then put in the appropriate global variable (**elbScalarValue** or **elbCompositeValue**) and three global flags are set as follows:

1.Expression can be evaluated and result is a 32 bit scalar value:
elbIsScalar32=True and result is in **elbScalarValue**

2. Expression can be evaluated and result is a 64 bit scalar value:
elbIsScalar64=True and result is in **elbCompositeValue**

2. Expression can be evaluated and result is a composite value:
elbIsComposite=True and result is in **elbCompositeValue**

The function returns True if the expression was evaluated, false otherwise.

The global variables are declared as externs in lvsexpr.h. To use this function, they must be declared in the application.

5. SCHEMA DEFINITION

This section defines the types, nodes and attributes of the intermediate format that are used to represent graphs. The highest node in the hierarchy is always the **control_flow_graph** node. This is attached to the *xtn_graph* attribute of different concurrent statements.

5.1 Primitive types

```
type  gphOperator is (
                        gphAbs,
                        gphAnd,
                        gphConcatenate,
                        gphDecrement,
                        gphDivide,
                        gphEqual,
                        gphExponent,
                        gphGreater_Than,
                        gphGreater_Than_Or_Equal,
                        gphIncrement,
                        gphLess_Than,
                        gphLess_Than_Or_Equal,
                        gphMinus,
                        gphMod,
                        gphMultiply,
                        gphNand,
                        gphNor,
                        gphNot,
                        gphNot_Equal,
                        gphOr,
                        gphPlus,
                        gphRem,
                        gphRol,
                        gphRor,
                        gphSla,
                        gphSll,
                        gphSra,
                        gphSrl,
                        gphXnor,
                        gphXor
);
```

5.2 Classes

5.2.1 GRAPH

A class covering all nodes constituting the **GRAPHGEN** domain.

5.2.2 CFG_BB_ITEM

A class containing the nodes that can be represented in a basic block partition of a CFG.

- **bb_transition**
- **cfg_basic_block**

5.2.3 CFG_CONDITION_EVALUATION

A class containing the nodes that can represent the result of a condition evaluation. If the condition is to be evaluated in the DFG, then the result of the evaluation will be placed on a dfg_cfg_interface edge. If it is to be evaluated in the CFG (a binary equality operation, for example) then the corresponding dfg will be empty and the condition is represented as a VIF expression of the class NAME_EXP. If there is no condition (loop…end loop;) the condition evaluation is empty.

- **dfg_cfg_interface**
- **NAME_EXP**

5.2.4 CFG_ITEM

A class containing all the nodes that can be represented in a CFG.

- **cfg_boolean_branch**
- **cfg_general_operation**
- **cfg_guarded_successor**
- **cfg_loop**
- **cfg_multiple_branch**
- **cfg_procedure_call**
- **cfg_wait**

5.2.5 CFG_PARTITION

A class representing the different possible partitions of a CFG.

- **control_flow_graph**
- **cfg_basic_block**
- **cfg_path**

5.2.6 CFG_PATH_ITEM

A class containing all nodes that can be represented in an execution path transition of a CFG.

- **cfg_path**
- **cfg_path_condition_test**

5.2.7 CFG_VIF_OPERATION

A class containing all VIF nodes that can be referenced through the cfg_vif_model attribute of CFG_ITEM nodes.

- **CONSTRAINT**
- **STATEMENT**
- **TOP_LEVEL_AGG**
- **cond_alternative**
- **for_iteration**

5.2.8 DFG_ADDRESS_DECODE

A class containing all DFG nodes that are used to evaluate the address of a non-scalar object type to be accessed including formal parameters.

- **dfg_allocated_address**
- **dfg_index_address**
- **dfg_parameter**
- **dfg_record_address**
- **dfg_segment**
- **dfg_slice_address**

5.2.9 DFG_CONNECTION

A class containing all DFG nodes representing a flow of data between two DFG nodes or a connection between the CFG and the DFG. If the result of a DFG operation is to be fed back to the CFG (a condition evaluation for example), then a dfg_cfg_interface edge is used.

- **dfg_cfg_interface**
- **dfg_data_edge**
- **dfg_merge_read**
- **dfg_merge_write**
- **dfg_select**

5.2.10 DFG_HARDWARE_RESOURCE

A class containing all nodes that can represent a hardware block in the synthesized result.

- **gph_functional_unit**

5.2.11 DFG_OPERAND

A class containing all DFG nodes that can represent a memory device. Because of VHDL's strong typing, addressing particular memories can be quite complex.

- **dfg_array_read**
- **dfg_array_write**
- **dfg_constant**
- **dfg_operand_read**
- **dfg_operand_write**

5.2.12 DFG_OPERATOR

A class containing all DFG nodes representing a functional operation.

- **dfg_abstract_operation**
- **dfg_abstract_timer**
- **dfg_function_call**
- **dfg_simple_operator_call**

5.2.13 DFG_VERTEX

A class containing all vertex nodes of a DFG.

- **Nested Class: DFG_ADDRESS_DECODE**
  - dfg_allocated_address
  - dfg_index_address
  - dfg_parameter
  - dfg_record_address
  - dfg_segment
  - dfg_slice_address
- **Nested Class: DFG_CONNECTION**
  - dfg_cfg_interface
  - dfg_data_edge
  - dfg_merge_read
  - dfg_merge_write
  - dfg_select
- **Nested Class: DFG_OPERAND**
  - dfg_array_read
  - dfg_array_write
  - dfg_constant
  - dfg_operand_read
  - dfg_operand_write
- **Nested Class: DFG_OPERATOR**
  - dfg_abstract_operation
  - dfg_abstract_timer
  - dfg_function_call
  - dfg_simple_operator_call

5.2.14 DFG_VIF_OPERATION

A class containing all VIF nodes that are referenced through the DFG.

- **ASSOCIATIONS**
- **FUNC_DEF**
- **NAME_EXP**
- **attribute_spec**
- **component_decl**
- **conversion_function_on_assoc**
- **gathered_partial_associations**
- **subtype_indicator**
- **type_conversion_on_assoc**
- **wait_stm**

5.2.15 DFG_VIF_DECLARATION

A class containing all VIF nodes that can be represented by an OPERAND node in a DFG.

- **FUNC_DEF**
- **OBJECT_ITEM**
- **PROC_DEF**

5.2.16 DFG_VIF_GRAPH_NODE

A class containing all VIF nodes that can possess a graph hierarchy.

- **concurrent_assertion_stm**
- **concurrent_proc_call_stm**
- **conditional_signal_assign_stm**
- **func_body**
- **proc_body**
- **process_stm**
- **selected_signal_assign_stm**

5.3 Nodes

5.3.1 bb_transition

A GRAPH node included in the CFG_BB_ITEM class. This node points to a basic block that can be a successor of the current basic block according to the flow of control. It can also indicate if the transition is clocked.
Contains the attributes:

cfg_bb_successor

A Required Alternate attribute that points to a cfg_basic_block node representing a basic block that can be executed immediately after the current basic block according to the flow of control.

cfg_event

An Optional Alternate attribute that points to a cfg_wait node. The cfg_wait node points to a wait_stm node in the VIF that contains the transition event.

5.3.2 cfg_basic_block

A GRAPH node included in the CFG_PARTITION and CFG_BB_ITEM classes. It represents a partition of the CFG in basic block format.
Contains the attributes:

cfg_item_s

A Required Alternate attribute that points to a list of nodes of the class CFG_ITEM that are included in the basic_block.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the basic block.

cfg_bb_transition_s

An Optional Alternate attribute that points to a list of bb_transition nodes representing pointers to the basic blocks that are conditionally executed immediately after the current basic block according to the flow of control.

5.3.3 cfg_boolean_branch

A GRAPH node included in the CFG_ITEM class. This node represents a conditional flow of control as modelled by the VHDL statements: "if", "exit" and "next".
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node. If the node represents an "if" statement, this attribute points to one of the cond_alternative attributes of the "if" statement rather than to the "if" statement itself. This is because an "if" statement generates a cfg_boolean_branch node for every condition in its list of alternatives (except the last one). For example, if we had an "if…elsif…elsif…else…end if" statement, three boolean_branch nodes will be generated.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the node.

cfg_true_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition evaluates to "true"

cfg_false_successor

An Optional Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition evaluates to "false". If an "if true…" statement is specified, this attribute will be null.

cfg_condition

A Required Alternate attribute that points to a node of the CFG_CONDITION_EVALUATION class representing the result of the evaluation of the condition.

5.3.4 cfg_general_operation

A GRAPH node included in the CFG_ITEM class. It models the general operations in VHDL (assignment, assertion, return,…).
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the node.

cfg_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node.

5.3.5 cfg_guarded_successor

A GRAPH node included in the CFG_ITEM class. This node contains a successor of a cfg_multiple_branch node representing a "case" statement as well as the value of the condition required for this successor to be taken.
Contains the attributes:

---

cfg_guarded_value

A Required Alternate attribute that points to a VIF node of the STATIC_VALUE class representing the value of the condition necessary for the successor to be taken

---

cfg_true_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition matches the guarded value.

---

cfg_break_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that represents the first node in the flow of control outside the case alternative region. In "C", this would be the first node executed after a break statement. Although this successor is not really part of the real control flow, it is useful to be easily able to identify it for many applications (partitioning into basic blocks for example).

5.3.6 cfg_loop

A GRAPH node included in the CFG_ITEM class. This node represents a conditional flow of control as modelled by the VHDL statements: "loop", "while…loop". It is assumed that "for" loops have either been unrolled or replaced by equivalent "while" loops.
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the node.

cfg_true_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition evaluates to "True"

cfg_false_successor

An Optional Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition evaluates to "False". If an infinite loop is specified, this attribute will be null.

cfg_condition

An Optional Alternate attribute that points to a node of the CFG_CONDITION_EVALUATION class representing the result of the evaluation of the condition. If there is no condition in the loop statement, this attribute will be null.

5.3.7 cfg_multiple_branch

A GRAPH node included in the CFG_ITEM class. This node represents a conditional flow of control as modelled by the VHDL "case" statement.
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the node.

cfg_guarded_successor_s

A Required Alternate attribute that points to a list of cfg_guarded_successor nodes that contain the successor node and the value of the "case" condition required for this successor to be taken.

cfg_condition

A Required Alternate attribute that points to a node of the CFG_CONDITION_EVALUATION class representing the result of the evaluation of the condition.

5.3.8 cfg_path

A GRAPH node included in the CFG_PARTITION and CFG_PATH_ITEM classes. It represents a partition of the CFG in execution path format.
Contains the attributes:

cfg_path_header

A Required Alternate attribute that points to node of the CFG_ITEM class that is the first node of the current execution path.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the path.

cfg_path_successor

A Required Alternate attribute that points to a cfg_path node representing an execution path that can be executed immediately after the current path according to the flow of control.

cfg_event

An Optional Alternate attribute that points to a wait_stm node in the VIF that contains the transition event.

5.3.9 cfg_path_condition_test

A GRAPH node included in the CFG_PATH_ITEM class. It represents one of the conditions that needs to be evaluated and the expected boolean value of the evaluation in order for the current transition to be taken. The transition will be taken if the logical "AND" of the evaluation of all of the cfg_path_condition_test nodes is "true".
Contains the attributes:

cfg_path_condition

A Required Alternate attribute that points to a node of the CFG_CONDITION_EVALUATION class representing the result of the evaluation of the condition.

cfg_path_cond_value

A Required Alternate attribute that points to a Boolean value that represents the expected result of the condition evaluation in order for the current path transition to be executed.

5.3.10 cfg_procedure_call

A GRAPH node included in the CFG_ITEM class. This node represents an unconditional flow of control that has the first node of a CFG representing the body of the procedure as successor. A second successor points to the node to be executed immediately after the end of the procedure.
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node.

dfg_parameter_s

A Required Alternate attribute that points to a list of dfg_parameter representing the evaluation of the subprogram parameters.

cfg_procedure_body

An Optional Alternate attribute that points to a control_flow_graph node representing the input to a CFG modelling the body of the procedure called.

cfg_successor

A Required Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node.

5.3.11 cfg_wait

A GRAPH node included in the CFG_ITEM class. It represents all possible variations of the "wait" statement.
Contains the attributes:

cfg_vif_model

A Required Alternate attribute that points to a VIF node of the CFG_VIF_OPERATION class representing the VIF model of this node.

dfg_model

An Optional Alternate attribute that points to a data_flow_graph node that is the point of entry to the DFG representing the node.

cfg_true_successor

An Optional Alternate attribute that points to a node of the CFG_ITEM class that is the successor of the current node if the condition evaluates to "true". If the condition is "false", the successor is the current node. If a simple "wait;" statement is specified, this attribute is null.

cfg_condition

An Optional Alternate attribute that points to a node of the CFG_CONDITION_EVALUATION class representing the result of the evaluation of the condition. If no condition is specified, this attribute is null.

dfg_timeout

An Optional Alternate attribute that points to a data_flow_graph node representing the evaluation of a timeout expression, if it exists in the original wait statement. If no time limit is specified, this attribute is null.

5.3.12 control_flow_graph

A GRAPH node included in the CFG_PARTITION class. It is attached to every concurrent statement of the VIF and contains the graph representation of that statement.
Contains the attributes:

cfg_item_s

A Required Primary attribute that points to a list of nodes of the class CFG_ITEM that constitute the CFG.

cfg_first_item

A Required Alternate attribute that points to a node of the class CFG_ITEM that is the first node of the CFG.

cfg_basic_block_s

An Optional Primary attribute that points to a list of cfg_basic_block nodes that represent the partition of the CFG into basic blocks.

cfg_first_basic_block

An Optional Alternate attribute that points to a cfg_basic_block node that is the first basic block of the CFG.

cfg_path_s

An Optional Primary attribute that points to a list of cfg_path nodes that represent the partition of the CFG into execution paths.

cfg_first_path

An Optional Alternate attribute that points to a cfg_path node that is the first execution path of the CFG.

5.3.13 data_flow_graph

A GRAPH node attached to every CFG and containing a representation of the data flow required to execute the corresponding CFG node.
Contains the attributes:

dfg_address_s

A Required Primary attribute containing a list of nodes of the class DFG_ADDRESS_DECODE present in the DFG. These nodes normally have no hardware equivalent.

dfg_connection_s

A Required Primary attribute containing a list of nodes of the class DFG_CONNECTION representing all of the edges of the DFG. Synthesis tools will map these operands onto connections.

dfg_operand_s

A Optional Primary attribute containing a list nodes of the DFG_OPERAND class that represent all of the operands of the DFG. Synthesis tools will map these operands onto memory devices.

dfg_operator_s

A Optional Primary attribute containing a list nodes of the DFG_OPERATOR class that represent all of the operations to be executed in the DFG. Synthesis tools will map these operands onto functional units.

5.3.14 dfg_abstract_operation

A GRAPH node belonging to the DFG_OPERATOR and DFG_VERTEX classes. It is used to model VHDL operations that may have no hardware equivalence (such as dynamic variable allocation (new BIT_VECTOR("0001")), file access operations, …) but possibly modify one or more operands.
Contains the attributes:

---

dfg_vif_model

A Required Alternate attribute that points to a node of the class DFG_VIF_OPERATION corresponding to a VIF node for which a DFG could not be generated.

---

dfg_functional_unit_s

An Optional Alternate attribute that points to a set of gph_functional_unit nodes. This attribute is filled by synthesis tools and contains a list of all functional units capable of executing this operation.

5.3.15 dfg_abstract_timer

A GRAPH node belonging to the DFG_OPERATOR and DFG_VERTEX classes.
Contains the attributes:

dfg_input_s

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX. These edges represent the data flow input to the operation.

dfg_functional_unit_s

An Optional Alternate attribute that points to a set of gph_functional_unit nodes. This attribute is filled by synthesis tools and contains a list of all functional units capable of executing this operation.

5.3.16 dfg_allocated_address

A GRAPH node belonging to the DFG_ADDRESS_DECODE and DFG_VERTEX classes. It is used to model the evaluation of an allocated memory access.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a node of the class DFG_VIF_OPERATION corresponding to a VIF "allocated" node.

dfg_pointer

An Optional Alternate attribute that points to a DFG_VERTEX node representing the evaluation of a prefixed name, if it exists.

dfg_declaration

A Required Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the array.

5.3.17 dfg_array_read

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to model an array read operation.
Contains the attributes:

dfg_index

A Required Alternate attribute that points to a node of the class DFG_VERTEX representing the evaluation of the address to be read.

dfg_declaration

A Required Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the array.

5.3.18 dfg_array_write

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to model an array write operation.
Contains the attributes:

dfg_input_s

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the values written.

dfg_index

A Required Alternate attribute that points to a node of the class DFG_VERTEX representing the address of the array to be written.

dfg_declaration

A Required Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the array.

5.3.19 dfg_cfg_interface

A GRAPH node belonging to the CFG_CONDITION_EVALUATION and the DFG_CONNECTION classes. It represents the data values that are passed from the DFG to the CFG, for example, the boolean result of a condition evaluation.
Contains the attributes:

dfg_dfg_side

An Optional Alternate attribute that points to a node of the DFG_VERTEX class corresponding to the generation of the value.

dfg_cfg_side

A Required Alternate attribute that points to a node of the CFG_ITEM class corresponding to the CFG node that requires the data represented by the edge.

5.3.20 dfg_constant

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to model a constant value.
Contains the attributes:

dfg_constant_value

A Required Alternate attribute that points to a node of the STATIC_VALUE (VIF) class corresponding to the value of the constant.

dfg_declaration

An Optional Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the constant value. If a literal is used, this attribute is null.

5.3.21 dfg_data_edge

A GRAPH node belonging to the DFG_CONNECTION class representing data transferred between two DFG nodes. If there is a delay imposed on the transfer of this data (i.e. an "after" clause is used) the dfg_delayed_exp attribute contains a sub-dfg corresponding to the evaluation of this delay . If the data is qualified or type converted, a pointer to a VIF "qualified" or "type_conversion" node is included.

Contains the attributes:

dfg_destination

A Required Alternate attribute that points to a node of the DFG_VERTEX class corresponding to the use of the value.

dfg_delayed_exp

An Optional Alternate attribute that points to a node of the DFG_VERTEX class corresponding to the evaluation of the delay.

dfg_conversion

An Optional Alternate attribute that points to a node of the VIF_CONVERSION class corresponding to the VIF node indicating the type of conversion or qualification to be executed on the data carried by the dfg_data_edge node.

5.3.22 dfg_function_call

A GRAPH node belonging to the DFG_OPERATOR and DFG_VERTEX classes. It represents any VHDL function call that does not correspond to a VHDL operator.
The inputs correspond to the function parameters. The output is the returned value.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a VIF node representing the declaration of the function called.

dfg_parameter_s

A Required Alternate attribute that points to a list of dfg_parameter representing the evaluation of the subprogram parameters.

dfg_output_s

An Optional Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the value returned by the function.

dfg_functional_unit_s

An Optional Alternate attribute that points to a set of gph_functional_unit nodes. If there is a functional unit in the library having the same name and the same profile as the function, it is added to this list automatically. Otherwise, this attribute will be filled by synthesis tools and will contain a list of all functional units capable of executing the function.

5.3.23 dfg_index_address

A GRAPH node belonging to the DFG_ADDRESS_DECODE and DFG_VERTEX classes and representing a single element memory access. Multi-dimensional array addresses are made up of series of this node, one for each dimension
Contains the attributes:

dfg_dimension

A Required Alternate attribute that points to a DFG_VERTEX node representing the result of the expression corresponding to the index address for a single dimension.

dfg_pointer

An Optional Alternate attribute that points to a DFG_VERTEX node representing the evaluation of a prefixed name, if it exists.

5.3.24 dfg_merge_read

A GRAPH node belonging to the DFG_CONNECTION class. This node is used to model the concatenation of a set of values read from a number of different memory locations. From a hardware point of view, it can be seen as the merging of a set of buses. If there is a delay imposed on the transfer of this data (i.e. an "after" clause is used) the dfg_delayedfg_exp attribute contains a sub-dfg corresponding to the evaluation of this delay . If the data is qualified or type converted, a pointer to a VIF "qualified" or "type_conversion" node is included.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a node of the class DFG_VIF_OPERATION representing the TOP_LEVEL_AGG node modelled.

dfg_delayed_exp

An Optional Alternate attribute that points to a node of the DFG_VERTEX class corresponding to the evaluation of the delay.

dfg_conversion

An Optional Alternate attribute that points to a node of the VIF_CONVERSION class corresponding to the VIF node indicating the type of conversion or qualification to be executed on the data carried by the dfg_data_edge node.

dfg_segment_s

A Required Alternate attribute that points to a list of dfg_segment nodes representing the different values to be merged and their corresponding addresses in the merged output.

5.3.25 dfg_merge_write

A GRAPH node belonging to the DFG_CONNECTION class. This node is used to model the concatenation of a set of input values to be written to a number of different memory locations. From a hardware point of view, it can be seen as the merging of a set of buses. If there is a delay imposed on the transfer of this data (i.e. an "after" clause is used) the dfg_delayedfg_exp attribute contains a sub-dfg corresponding to the evaluation of this delay . If the data is qualified or type converted, a pointer to a VIF "qualified" or "type_conversion" node is included.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a node of the class DFG_VIF_OPERATION representing the TOP_LEVEL_AGG node modelled.

dfg_delayed_exp

An Optional Alternate attribute that points to a node of the DFG_VERTEX class corresponding to the evaluation of the delay.

dfg_conversion

An Optional Alternate attribute that points to a node of the VIF_CONVERSION class corresponding to the VIF node indicating the type of conversion or qualification to be executed on the data carried by the dfg_data_edge node.

dfg_segment_s

A Required Alternate attribute that points to a list of dfg_segment nodes representing the different values to be merged and their corresponding addresses in the merged output.

dfg_input_s

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the memory locations to be modified.

5.3.26 dfg_operand_read

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to model the reading of an operand.
Contains the attributes:

---

dfg_declaration

A Required Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the array.

5.3.27 dfg_operand_write

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to model the assignment of an operand.
Contains the attributes:

dfg_input_s

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the values written.

dfg_declaration

A Required Alternate attribute that points to a node of the OBJECT_ITEM (VIF) class corresponding to the declaration of the array.

5.3.28 dfg_parameter

A GRAPH node belonging to the DFG_ADDRESS_DECODE and DFG_VERTEX classes. It represents the mapping of the actual parameter used in a function call to the corresponding formal parameter.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a DFG_VIF_OPERATION node of the class ASSOCIATIONS representing the VIF model of the parameter mapping.

dfg_actual

A Required Alternate attribute that points to a DFG_VERTEX representing the evaluation of the actual part of the function parameter.

dfg_formal

A Required Alternate attribute that points to a DFG_VERTEX representing the evaluation of the formal part of the function parameter.

dfg_mode

A Required Alternate attribute that points to an Interface_Element_Mode value representing the direction of the parameter.

5.3.29 dfg_record_address

A GRAPH node belonging to the DFG_ADDRESS_DECODE and DFG_VERTEX classes. This node is used to model the evaluation of a memory address corresponding to an element of a record.
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a node of the class DFG_VIF_OPERATION that represents the corresponding VIF "indexed" node.

dfg_pointer

An Optional Alternate attribute that points to a DFG_VERTEX node representing the evaluation of a prefixed name, if it exists.

dfg_field

A Required Alternate attribute that points to an element_decl node corresponding to the declaration of the particular record field.

5.3.30 dfg_segment

A GRAPH node belonging to the DFG_OPERAND and DFG_VERTEX classes. It is used to represent the value of part of a concatenated bus and the bus bounds containing this value. Contains the attributes:

dfg_value

A Required Alternate attribute that points to a node of the class DFG_CONNECTION representing the value to be merged.

dfg_bound

A Required Alternate attribute that points to a DFG_ADDRESS_DECODE node that gives the left and right bounds of the current bus segment, or a simple index if only one element is accessed.

5.3.31 dfg_select

A GRAPH node belonging to the DFG_CONNECTION class. It is used to model "case" and "if…elsif…else" statements in pure data flow.
Contains the attributes:

dfg_select_condition

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the result of the condition evaluation.

dfg_input_s

A Required Alternate attribute that points to a list of nodes of the class DFG_VERTEX representing the input values to the node.

dfg_functional_unit_s

An Optional Alternate attribute that points to a set of gph_functional_unit nodes. This attribute is filled by synthesis tools and contains a list of all functional units capable of executing this operation.

5.3.32 dfg_simple_operator_call

A GRAPH node belonging to the DFG_OPERATOR and DFG_VERTEX classes and representing all monadic and dyadic synthesizable VHDL operations.
Contains the attributes:

dfg_vif_model

An Optional Alternate attribute that points to a node of the class DFG_VIF_OPERATION corresponding to the VIF representation of the operation to be executed.

dfg_operation

An Optional Alternate attribute that points to a value of type ggOperator representing the type of operator to be executed. In addition to VHDL's built-in operators, the graph generation recognises operations of type increment and decrement.

dfg_input_left

A Required Alternate attribute that points to a node of the class DFG_VERTEX representing the left hand side value for the operation. For monadic operations, this is the only input required.

dfg_input_right

An Optional Alternate attribute that points to a node of the class DFG_VERTEX representing the right hand side value for the operation.

dfg_functional_unit_s

An Optional Alternate attribute that points to a set of gph_functional_unit nodes. This attribute is filled by synthesis tools and contains a list of all functional units capable of executing this operation.

5.3.33 dfg_slice_address

A GRAPH node belonging to the DFG_ADDRESS_DECODE and DFG_VERTEX classes. This node is used to model the evaluation of a slice of memory addresses .
Contains the attributes:

dfg_vif_model

A Required Alternate attribute that points to a DFG_VIF_OPERATION node corresponding to a VIF "slice" node representing the set of addresses to be accessed.

dfg_pointer

An Optional Alternate attribute that points to a DFG_VERTEX node representing the evaluation of a prefixed name, if it exists.

dfg_static_bounds

A Required Alternate attribute that points to a VIF_BOOLEAN value that indicates whether or not the bounds are static. If either of them is dynamic it will contain a dfg_data_edge that points to a sub-DFG representing the evaluation of the bound. In this case, the attribute will be False. If both bounds are static, they will point to dfg_constant nodes containing the bound values and this attribute will be True.

dfg_left_bound

An Optional Alternate attribute that points to a node of the class DFG_VERTEX which represents the evaluation of the left-most address in the range of addresses. If the address cannot be statically evaluated, this attribute will be null.

dfg_right_bound

An Optional Alternate attribute that points to a node of the class DFG_VERTEX which represents the evaluation of the right-most address in the range of addresses. If the address cannot be statically evaluated, this attribute will be null.

5.3.34 gph_functional_unit

A GRAPH node belonging to the DFG_HARDWARE_RESOURCE class representing a functional unit declared in the ABSYNT library.
Contains the attributes:

gph_fu_declaration

A Required Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the VIF representation of the FU declaration.

gph_fu_area

An Optional Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the FU_AREA attribute of the functional unit defining its area if the attribute was specified.

gph_fu_execution_time

An Optional Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the FU_EXECUTION_TIME attribute of the functional unit defining the execution time if the attribute was specified.

gph_fu_power

An Optional Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the FU_POWER attribute of the functional unit defining the power requirements if the attribute was specified.

gph_fu_fan_in

An Optional Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the FU_FAN_IN attribute of the functional unit defining the fan-in if the attribute was specified.

gph_fu_fan_out

An Optional Alternate attribute that points to a DFG_VIF_OPERATION node that in turn points to the FU_FAN_OUT attribute of the functional unit defining the fan-out if the attribute was specified.

6. APPENDIX A: LPIKEY

This appendix explains the utilisation of lpikey to obtain authorization to use any LVS product.

6.1 Key Management using “lpikey”

The command **lpikey** sets the authorization keys for the LPI extensions such as APEX, GraphGen, GEME or VELS. This command can be found in the directory **./Utils** of any LPI extension installation diskette. For more information about the different LPI extensions proposed by LEDA, please send an E-mail to `sales@leda.fr`.

There are two ways to authorize the LPI extension(s) within a user’s application: by validating the LPI kernel archive for the given LPI extension(s), or by validating the executable built by the user with the LPI kernel archive. These two modes are described in the next two sections.

The authorization key mechanism used by **lpikey** is totally compatible with the authorization key mechanism used by the LVS Compiler when executing the command “v key” (please refer to the section 6.27 “key” of the manual “LEDA VHDL System - User’s Manual” and to the section 2 “LVS Installation” of the manual “LEDA VHDL System - Implementor’s Guide Part I”).

Warning: in order to be able to execute the command **lpikey**, the user must have write permission on the file **lvskernel.a** (first mode) or the executable (second mode), as well as write permission in the directory where this archive or executable is located.

6.1.1 Validation of the LPI kernel archive (lvskernel.a)

Any user application must be linked with the LPI kernel archive file **lvskernel.a**. This archive file is given in the directory **./lvskern** of the installation diskette of LVS. To validate an application including one or more LPI extension(s), it is sufficient and recommended to validate **lvskernel.a** with the authorization key(s) provided by LEDA. Entering a given authorization key in the archive file is done by executing the following command:

```
lpikey <key>
```

where `<key>` is the authorization key provided by LEDA for the given LPI extension(s). When this command line is executed, the **lpikey** program looks for the file **lvskernel.a** in the following directory path:

```
. ./lvskern $LVS_PATH/lvskern /usr/local/lvs/lvskern
```

and updates the first found with the corresponding key. Then, all executables built by the user with the validated archive file will have the authorizations corresponding to the key(s) entered in this archive file.

6.1.2 Validation of the user’s executable

The user can also enter an authorization key directly into the executable of his or her application. This is necessary when the user receives from LEDA an evaluation copy of an LPI extension (usually provided as an executable to facilitate the user’s installation), or when the user wants to execute his or her executable on a different host to the one for which the LPI kernel archive has been validated. The following command must be executed to validate a given executable:


```
lpikey -f <executable> <key>
```

where <executable> is the name of the user's executable and <key> is the authorization key provided by LEDA for the given LPI extension(s). If this command line is executed, then the provided authorization key is entered into the executable itself of the user's application.

If the LPI kernel archive does not contain any authorization key valid for a given LPI extension, then the command **lpikey** must be executed for each executable linked with this LPI kernel archive that requires the given LPI extension, and this must be done each time the executable is built. To avoid this, it is recommended to validate the archive file itself, if possible.

If the LPI kernel archive already contains a certain number of authorization keys, then the command **lpikey** above will add another authorization key to the executable, which also inherits all the authorization keys from the archive file.

6.1.3 Revalidating the same authorization key(s)

It may sometimes be necessary to revalidate the same authorization keys. This happens when the user receives a new release of the LVS system, in which case the user needs to enter his or her current authorization keys into the new version of the LPI kernel archive file **lvskernel.a**. Another example is when the user does not enter any authorization key in the LPI kernel archive but only in the user's executable itself: the authorization key must be reentered in the executable each time it is rebuilt (see previous paragraph 1.2).

To revalidate the same authorization keys, the user just needs to execute:

```
lpikey
```

to validate the new LPI kernel archive file **lvskernel.a** (located into the same directory path as the one mentioned above) with the current authorization keys, and the command:

```
lpikey -f <executable>
```

to validate the executable whose name is provided.

The current authorization keys are stored by **lpikey** in a file named **vLastKey** which is located in the same directory as the one containing the authorized archive file or executable. The command **lpikey** must be executed at least once by the user with an explicit key in order to create the file **vLastKey**. This file is then automatically updated at each execution of the command.

6.1.4 Checking the authorization key(s)

In order to check the current authorization keys, the user can execute the command:

```
lpikey list
```

to check the authorization keys of the LPI kernel archive file **lvskernel.a** (located in the same directory path as the one mentioned above), and the command:

```
lpikey -f <executable> list
```


to check the authorization keys of the executable whose name is provided (the listed authorization keys also includes those of the kernel archive that have been inherited).

This command has the same behavior as the command “v key list” of the LVS Compiler.

6.1.5 Help

The command **lpikey** includes a help which can be displayed by executing:

```
lpikey -h
```

The following message is then displayed:

```
Command lpikey sets the authorization key(s) for LPI extensions:

       lpikey [-f <executable>] [<key> | list]

With option -f, lpikey executes on the program <executable>
Without option -f, lpikey executes on the archive lvskernel.a
found in path (. ./lvskern $LVS_PATH/lvskern /usr/local/lvs/lvskern)

If <key> is provided, then <key> is used as the authorization key
If list is provided, then lpikey lists all stored authorization keys
Otherwise, lpikey uses authorization key(s) of local file vLastKey

To get this help again, you can execute:

       lpikey -h | -help
```

