Information extraction and validation of CDFG in 
NoGap

Examensarbete utfört i Datorteknik 
vid Tekniska högskolan vid Linköpings universitet

av

Mónica Sánchez Yagüe

LiTH-ISY/ERASMUS-A–13/003–SE

Linköping 2013
Information extraction and validation of CDFG in NoGap

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Mónica Sánchez Yagüe

LiTH-ISY/ERASMUS-A–13/003–SE

Handledare: Per Karlström
ISY, Linköpings universitet

Examinator: Per Karlström
ISY, Linköpings universitet

Linköping, 27 maj 2013
Information extraction and validation of CDFG in NoGap

Mónica Sánchez Yagüe

A Control Data Flow Graph (CDFG) is a Directed Acyclic Graph (DAG) in which a node can be either an operation node or a control node. The target of this kind of graph is to capture all the control and data flow information of the original hardware description while preserving the various dependencies.

This kind of graph is generated by Novel Generator of Accelerators and Processors (NoGap), a design automation tool for Application Specific Instruction-set Processor (ASIP) and accelerator design developed by Per Karlström from the Department of Electrical Engineering of Linköping’s University.

The aim of this project is to validate the graph, check if it fulfills the requirements of its definition. If it does not, it is considered an error and the running process will be aborted. Moreover, useful information will be extracted from the graph for future work.
Abstract

A Control Data Flow Graph (CDFG) is a Directed Acyclic Graph (DAG) in which a node can be either an operation node or a control node. The target of this kind of graph is to capture all the control and data flow information of the original hardware description while preserving the various dependencies.

This kind of graph is generated by Novel Generator of Accelerators and Processors (NoGap), a design automation tool for Application Specific Instruction-set Processor (ASIP) and accelerator design developed by Per Karlström from the Department of Electrical Engineering of Linköping’s University.

The aim of this project is to validate the graph, check if it fulfills the requirements of its definition. If it does not, it is considered an error and the running process will be aborted. Moreover, useful information will be extracted from the graph for future work.
Acknowledgments

First of all, I would like to thank Per Karlström for letting me do my final project with him. Even though he stopped working at the university, he continued being my supervisor.

Moreover, I want to thank my parents and my sister because they have always been there for me, even in the distance. And finally, I want to thank my new friends in Sweden and the ones in Spain for all their support in the good and bad moments.

Ett stort tack till alla.

Linköping, Maj 2013
Mónica Sánchez Yagüe
## Contents

List of Figures ix

Abbreviations xiii

1 Background knowledge 1
   1.1 C++ 1
   1.2 NoGap 1
   1.3 Graphs 3
       1.3.1 Graphs in NoGap 3

2 Introduction 5
   2.1 Control Data Flow Graph 5
       2.1.1 Vertices 5
       2.1.2 Edges 6
   2.2 Validation of the CDFG 7

3 Obtaining the CDFG 9
   3.1 Generator System 9
   3.2 Extraction of the CDFG 9

4 Search of cycles in a CDFG 11
   4.1 Remove cycle assign edges 11
   4.2 Cycle detection algorithm 12

5 Validation of inputs and outputs 15
   5.1 Extraction of inputs and outputs 15
   5.2 Validation of inputs and outputs 16

6 Generation of C++ code 19
   6.1 Generator System 19
   6.2 Find ports and operations 19
       6.2.1 Generating combinational assignments 22
       6.2.2 Generating cycle assignments 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.3</td>
<td>Generating switch statements</td>
<td>24</td>
</tr>
<tr>
<td>6.3</td>
<td>Generate C++ code</td>
<td>24</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Example of code generated</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Information extraction</td>
<td>31</td>
</tr>
<tr>
<td>7.1</td>
<td>Search for all paths</td>
<td>31</td>
</tr>
<tr>
<td>7.2</td>
<td>Show all paths</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions and future work</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Example of a CDFG .............................................. 7
6.1 CDFG ............................................................. 27
List of Listings

1.1 Example of an FU specification ........................................ 2
1.2 BGL graph definition .................................................... 3
2.1 Description of CDFG ..................................................... 6
3.1 Method to process the CDFG ............................................ 10
4.1 Structure for the filtered edges ....................................... 12
4.2 Visitor for detecting cycles ............................................ 13
4.3 Method for searching cycles ........................................... 13
5.1 Extraction of inputs and outputs ..................................... 15
5.2 Method for validating inputs and outputs .......................... 16
6.1 Class for generating C++ code ........................................ 20
6.2 Method to find ports .................................................... 20
6.3 Method to find operations ............................................. 21
6.4 Method to generate combinational assignments .................. 23
6.5 Method to generate cycle assignments ............................... 24
6.6 Function to generate C++ code ....................................... 25
6.7 FU specification .......................................................... 26
6.8 test.hh ........................................................................... 28
6.9 test.cpp ......................................................................... 29
7.1 Use of BFS ................................................................. 32
7.2 Visitor for BFS ............................................................. 32
7.3 Call to the show_path method ......................................... 33
7.4 Method to show all paths ............................................... 34
Abbreviations

ASIP  Application Specific Instruction-set Processor
BFS  Breadth First Search
BGL  Boost Graph Library
CDFG  Control Data Flow Graph
DAG  Directed Acyclic Graph
DFS  Depth First Search
EDA  Electronic Design Automation
FF  Flip-Flop
FU  Functional Unit
HDL  Hardware Description Language
HW  Hardware
IR  Intermediate Representation
CL  Common Language
Novel Generator of Accelerators and Processors
PU  Parse Unit
RTL  Register Transfer Language
STL  Standard Template Library
VHDL  VHSIC Hardware Description Language
VHSIC  Very-High-Speed Integrated Circuit
Abbreviations
Before starting with this project, I have learnt some basic tools that I have used for the development of this thesis. Therefore, the reader should have a certain level of understanding in these topics. The most important ones are described in this chapter.

1.1 C++

A good knowledge of C++ [6] is required since the majority of the project has been programmed in this language, including the recent version C++11 [7]. Concepts such as pointers, structures, functions, namespaces, templates, classes, inheritance and polymorphism should be well understood. Moreover, it is necessary to be familiar with the Standard Template Library (STL) [6] since it is very often used during this thesis.

1.2 NoGap

There are two different ways of designing a processor or a pipelined datapath. On one hand, a Hardware Description Language (HDL) such as Verilog or VHSIC Hardware Description Language (VHDL) offers a low level control at the register transfer level. However, it has the disadvantage that some small details like hardware multiplexing, instruction decoding, control signal generation and control signal pipelining need to be managed. On the other hand, a high level design tool, often called Electronic Design Automation (EDA) tool, offers an easy construction of instruction controlled datapaths while losing control over the final hardware. NoGap [4] is a design automation tool for ASIP and accelerator design developed
by Per Karlström from the Department of Electrical Engineering of Linköping’s University. It offers the advantages of both design methods and solves the disadvantages of each one without choosing between a low or high level tool.

It allows humans the simple tasks like designing individual modules and specifying the temporal and spatial relations between these modules on a per instruction basis. Then NoGap is responsible for merging all these instructions into a pipelined ASIP architecture having the necessary control signals, multiplexers and associated delays. Therefore, one of the design inputs to NoGap is the description of the individual Functional Units (FUs) that are interconnected in some kind of network. The general principle of NoGap is based upon this interconnection.

NoGap Common Language (NoGapCL) is a flexible language created for NoGap and used for hardware description. In particular, the description of the FUs is done in this language which is a mixture of VHDL, Verilog and C, so the reader should be familiar with these languages. An example of an FU specification can be seen in Listing 1.1.

```
Listing 1.1: Example of an FU specification

fu test
{
  input a_i;
  input b_i;
  input c_i;
  output res_o;
  signal s_comb;
  signal s_cycle;
  cycle
  {
    switch(a_i)
    {
      0: %NOP. default{}
      1: %WRITE
      {
        s_cycle = a_i | b_i;
      }
    }
  }
  comb
  {
    if(b_i == 0)
    {
      s_comb = c_i;
    }
    else
    {
      s_comb = b_i;
    }
  }
  comb
  {
    res_o = s_cycle ^ s_comb;
  }
}
```
1.3 Graphs

Graphs are one of the basis of this thesis, thus the reader should have knowledge of graph theory like vertices, edges and cycles as well as their types and properties. Moreover, some graph algorithms will be used for the development of this project like Depth First Search (DFS), Breadth First Search (BFS) or topological sort, but they will be explained in the next chapters.

1.3.1 Graphs in NoGap

All graphs in NoGap use the Boost Graph Library (BGL) [2]. The way to handle graphs may seem similar to containers but it is more complicated due to, for example, the different ways that graph algorithms can traverse a graph.

The `adjacency_list` class is a template from the BGL where the data structure used to represent the out-edges for each vertex and the vertex set are given as parameters. Moreover, the type of graph can be chosen between bidirectional, directed or undirected depending the access required for out-edges and in-edges. `NodeData` and `EdgeData` represent the data associated with the vertices and edges in the graph, being necessary class hierarchies when there are a number of different types. Listing 1.2 shows how to define a BGL graph.

```
Listing 1.2: BGL graph definition

typedef boost::adjacency_list<boost::vecS,
  boost::vecS,
  boost::bidirectionalS,
  NodeData,
  EdgeData,
  boost::vecS> graph;
```

A quick tour of BGL can be found in [2] in order to understand more concepts that will be used during this thesis.
2.1 Control Data Flow Graph

A CDFG [5] is a DAG in which a node can be either an operation node or a control node. The target of this kind of graph is to capture all the control and data flow information of the original hardware description while preserving the various dependencies. Therefore, the directed edges show the transfer of a value or control between nodes.

The CDFG is implemented as the class arch_data::cdfg::Graph, which is composed of, as its most important member variable, a BGL graph of type arch_data::cdfg::cdfg_t, as seen in Listing 2.1. The definition of cdfg_t shows that for the vertices and edges data a boost shared pointer is used.

It should be pointed out that all logic written in NoGapCL takes place within either a cycle or a comb block. The comb block describes combinational logic and it means that the connections have no delay. On the other hand, the cycle block describes synchronous logic which means that the signals are registered through one or more Flip-Flops (FFs).

2.1.1 Vertices

There are a number of different vertices types and each type is described as a class that derives from the arch_data::cdfg::vertex::Vertex class. In the graph, each vertex type has its own color, shape and name so that they can be differentiated.

The CycleAssign, CombAssign and Condition vertices represent the different cycle, comb and condition blocks. On the other hand, the BlockStart and BlockEnd
mark the start and the end of comb, cycle or condition blocks. Finally, the Symbol vertices are the inputs, outputs and signals of the graph.

### 2.1.2 Edges

In the case of edges, there are also a number of different types which are distinguished by their color and name. The class that describes each type inherits from `arch_data::cdfg::edge::Edge`.

Operand edges (Op) go from operands to the blocks where they are read which can be either a comb or cycle block. Assignment edges go from a block to the variables being assigned in that block. These assignment edges can be of two types, cycle assign (Cy=) or combinational assign (Co=). Cycle assign edges are used if a variable is assigned in a cycle block and thus forms a register. Combinational assign edges are used if a variable is assigned in a comb block and thus forms combinational logic.

Condition edges connect a BlockStart with the block where the condition is defined and CaseCondition edges connect the condition block with the assign vertex. TrueFlow and FalseFlow go from a condition vertex depending on whether the condition is true or not. Flow edges end in a BlockEnd but can start from another BlockEnd, a BlockStart or an assign vertex. Finally, BlockPair edges are the ones which connect each BlockStart with its BlockEnd.

An example of a CDFG can be seen in Figure 2.1 and it refers to the FU specification described in Listing 1.1.

---

**Listing 2.1: Description of CDFG**

```cpp
namespace arch_data
{
namespace cdfg
{
    typedef boost::shared_ptr<vertex::Vertex> node_info_t;
    typedef boost::shared_ptr<edge::Edge> edge_info_t;
    typedef boost::adjacency_list<boost::listS, boost::listS, boost::bidirectionalS, node_info_t, edge_info_t, boost::listS> cdfg_t;
    ...

    class Graph: public boost::noncopyable
    {
        ...
        const cdfg_t& graph_m;
        ...
    };
}
}
```

---
2.2 Validation of the CDFG

Starting from the hardware description, NoGap generates a CDFG. It is, however, necessary to check for possible errors in the graph which means that it has to fulfill all the requirements below. They will be explained and implemented in the next chapters.

- Remove the cycle assign edges and check that the graph has no cycles: In order to synthesize cycle based software simulators, combinational cycles should be removed.
- Validate the inputs and outputs: Inputs cannot have in-coming edges and outputs cannot have out-going edges.
- Generate C++ code: Translate the parallel execution of the graph into sequential code.

Therefore, the aim of this project is to validate these requirements and if there is an error, the running process will be aborted. Moreover, useful information will be extracted from the graph in order to be used in future work.
3 Obtaining the CDFG

3.1 Generator System

Firstly, it is necessary to point out the concept of Parse Unit (PU) which is a very important concept in NoGap and contains all relevant Intermediate Representation (IR) for one FU. It is implemented as a PU class with a number of different member variables used for the IR, one of the variables is the proper CDFG.

All the classes, structures and functions required to accomplish the aim of this thesis are managed by NoGap using a generator system. A generator is a software component that processes PUs. It is implemented as a C++ class derived from the generator::Generator virtual base class. For more information about generators or PU, see [4].

The classes used in this thesis inherit from the generator::Generator virtual class cited before which has two methods that are overridden as it can be seen in Listing 3.1. The first one is the generate_predicate() method which inspects the PU in the argument and returns true if the PU should be processed by the generator. In this case, it returns if the control_data_flow_graph_m variable exists which means that there is a CDFG to be processed. The other one is the operator() method which performs the actual generation. All future functions discussed in this thesis are called from this method.

3.2 Extraction of the CDFG

The first task to do before starting with the validation of the CDFG is actually obtain it. This is done calling to the function graph() with the PU object which has
access to the control_data_flow_graph_m variable. It returns a reference to the proper graph as it can be seen below.

```cpp
const cdfg_t& graph = pu.control_data_flow_graph_m->graph();
```

Now that the CDFG has been extracted, the next step is to deal with it in order to validate and obtain useful information.
Search of cycles in a CDFG

The first task to do in the CDFG is checking if there are cycles or not. This is done first by removing the cycle assign edges (Cy=) in the CDFG and then running a standard cycle detection algorithm. However, it is possible to write code that describes combinational cycles but it is not advisable when designing at the Register Transfer Language (RTL) and it would make impossible NoGap from synthesize cycle based software simulators. Moreover, it does not accord with the definition of DAG.

4.1 Remove cycle assign edges

A CDFG can have cycle assign edges but they should be removed before searching for combinational cycles.

The filtered_graph class template from the BGL [2] creates a filtered view of a graph, allowing to filter the required edges and vertices. This template just need to know the graph to be filtered and the edge predicate and vertex predicate needed for choosing the right edges and vertices. In this case, the cycle assign edges are the only ones needed to be filtered so the structure shown in Listing 4.1 is used as the edge predicate. It has a function operator() which is responsible for extracting the edge type and returning the non-filtered edges.

The way to use this structure is shown below:

```cpp
boost::filtered_graph<cdfg_t, filtered_edges> fg(graph, filtered_edges(graph));
```
4.2 Cycle detection algorithm

According to the BGL [2], DFS is an algorithm that can be used to detect cycles and it is implemented in this library.

DFS [3] consists on visiting all the vertices in a graph going always deeper, which means that it will explore out-edges of the last discovered vertex that still have unexplored edges leaving it. Then, it will backtrack to the previous vertex and continue with the unexplored edges from that vertex. When all the reachable vertices from a particular source vertex have been discovered, if there are undiscovered vertex remain, then one of them is chosen as the source vertex and the search continues. The algorithm will finish when all the vertices have been discovered. This process creates a set of depth-first trees which together form the depth-first forest.

The depth_first_search() function performs a depth-first traversal of the vertices in a directed graph. This function can be adapted through a visitor which is a mechanism that gives the user control to extend the functionality of the graph search algorithms.

This mechanism can be seen in Listing 4.2 as an structure that inherits from dfs_visitor<> which means that it provides the visitor with empty versions of the DFSVisitor methods. Then it is only necessary to define the methods that are going to be used, in this case back_edge() and tree_edge(). The back_edge() method is called when DFS explores an edge to an already discovered vertex. If it happens, it means that there is a cycle so the has_cycle_m variable is updated to true and the edge is stored. The tree_edge() method is invoked on each edge as it becomes a member of the edges that form the search tree. In this case, it will be used to print all the edges of a cycle when there is one.

An object of the cycle_detector structure is passed by value to the depth_first_search() function in Listing 4.3 together with the filtered graph and a color map that is used to keep track of the search progress through the graph. When the search finishes, the has_cycle variable has the information about if the graph has cycles or not. If there are, it is an error and the running
### Listing 4.2: Visitor for detecting cycles

```cpp
struct cycle_detector : public boost::dfs_visitor<>
{
  protected:
    bool has_cycle_m;
    std::vector<edesc_t>& back_edges_m;
    std::vector<edesc_t>& tree_edges_m;
  public:
    cycle_detector(bool has_cycle_, std::vector<edesc_t>& back_edges_,
                   std::vector<edesc_t>& tree_edges_ ) : has_cycle_m(has_cycle_),
               back_edges_m(back_edges_), tree_edges_m(tree_edges_) {} 
    template <class Edge, class Graph>
    void tree_edge(Edge e, Graph&); 
    template <class Edge, class Graph>
    void back_edge(Edge e, Graph&); 
};
```

Process is aborted. For each back edge, an error message will be shown with the edges that forms the cycle. This path is printed with the `show_path()` method that is shown in Listing 7.4 and explained in that chapter.

### Listing 4.3: Method for searching cycles

```cpp
void ProcessControlDataFlowGraph::graph_has_cycle(const cdfg_t& g) const
{
  bool has_cycle = false;
  std::vector<edesc_t> back_edges;
  std::vector<edesc_t> tree_edges;
  cycle_detector vis(has_cycle,back_edges,tree_edges);
  utils::vertex_index_map_factory<cdfg_t> vertex_index_map(g);
  utils::color_map_factory<cdfg_t> color_map(g,vertex_index_map);
  boost::depth_first_search(fg,vis,color_map.get());
  if(has_cycle)
  {
    for(const auto& index: back_edges)
    {
      std::stringstream ss;
      ss << show_path(tree_edges, g, boost::target(index,g), boost::source(index,g));
      using namespace nogap_system;
      compile_issue(Issue::severity_t::ERROR) << "Cycle detected: " << ss.str();
    }
  }
}
```

Finally, if there are no cycles after removing the cycle assign edges, the CDFG fulfill the first requirement, so it is possible to continue checking the other ones in the next chapters.
Validation of inputs and outputs

Inputs cannot have in-coming edges and outputs cannot have out-going edges. It seems obvious but it is necessary to check this property in the generated CDFG.

5.1 Extraction of inputs and outputs

As it was explained before, there are a number of different types of vertices in a CDFG. Symbol vertices represent the inputs, outputs and signals, so it is necessary to distinguish them.

The symbol table [4] is a hierarchical table of identifiers which is used to search and validate symbols during the compilation process. It is implemented as the class parser::SymbolTable and contains parser::Symbol objects. Listing 5.1 shows the use of the parser::SymbolCollection type which is just a vector of Symbols. There are two parser::SymbolCollection defined which are given as a parameter to the parser::in_out_ports() function together with the PU object. The function returns the two parser::SymbolCollection, one with the inputs and the other one with the outputs of the graph. This way the signals have been dismissed and it is possible to operate just with the required vertices.

Listing 5.1: Extraction of inputs and outputs

```cpp
1 parser::SymbolCollection in_ports, out_ports;
2 parser::in_out_ports(pu, in_ports, out_ports);
```
5.2 Validation of inputs and outputs

The next step is to check that inputs do not have in-coming edges and outputs do not have out-going edges. Listing 5.2 shows how it is done. For each port (input or output), it is necessary to find its vertex in the CDFG. Therefore, the name of the vertex is compared to the name of the port. Moreover, the color of the vertex is also checked due to some errors in the algorithm when there are switch statements. Once the vertex has been found, it is stored in a vector that will be returned at the end of the function in order to have a vector of inputs vertices and another one of outputs vertices.

Listing 5.2: Method for validating inputs and outputs

```cpp
std::vector<vdesc_t> ProcessControlDataFlowGraph::validate_inputs_outputs
(const cdfg_t& g, parser::SymbolCollection& ports, bool input) const
{
    std::stringstream ss;
    std::vector<vdesc_t> vec;
    for (const auto& elem : ports)
    {
        for (auto vp = boost::vertices(g); vp.first != vp.second; ++vp.first)
        {
            vertex::Vertex& v = *g[*vp.first];
            if (v.text() == elem->name() && v.color() == "gray")
            {
                vec.push_back(*vp.first);
                if (input)
                {
                    auto pair_iter = boost::in_edges(*vp.first, g);
                    for (auto iter = pair_iter.first; iter != pair_iter.second; ++iter)
                    {
                        const edge_info_t& edge_info = g[*iter];
                        const std::type_info& ti = typeid(*edge_info);
                        ss << elem->name() << ", Edge->" << ti.name();
                        using namespace nogap_system;
                        compile_issue(Issue::severity_t::ERROR) << "Incoming-edge for input: " << ss.str();
                    }
                }
                else
                {
                    auto pair_iter = boost::out_edges(*vp.first, g);
                    for (auto iter = pair_iter.first; iter != pair_iter.second; ++iter)
                    {
                        const edge_info_t& edge_info = g[*iter];
                        const std::type_info& ti = typeid(*edge_info);
                        ss << elem->name() << ", Edge->" << ti.name();
                        using namespace nogap_system;
                        compile_issue(Issue::severity_t::ERROR) << "Outcoming-edge for output: " << ss.str();
                    }
                }
            }
        }
    }
    return vec;
}
```

It is necessary to distinguish inputs from outputs and it is done by a boolean that is given as a parameter to the function. If the port that is being analyzed is an
input and it has an in-coming edge the compilation process will finish and the non-valid input will be displayed together with the in-coming edge. If it is an output, the same process is done but with its out-going edges.

The way this function is used is shown below.

```cpp
auto inputs = validate_inputs_outputs(graph, in_ports, true);
auto outputs = validate_inputs_outputs(graph, out_ports, false);
```

There are two calls to the function in order to have a vector with the inputs and another one with the outputs that will be used in the next chapters.
6

Generation of C++ code

Despite each block is a sequential execution description, the execution between blocks is parallel and needs to be sequentialized. This process is done analyzing the graph and the vertices in a particular order and then generating C++ code from it.

6.1 Generator System

For the development of this task a generator system is also used as a C++ class derived from the generator::Generator virtual base class. In this case, apart from the methods generate_predicate() and operator(), there are some member variables declared that will be used later. These are the name of each input, output and signal which will be stored in a vector of strings as well as the operation vertices which will be stored in a vector of vdesc_t for each operation type. Moreover, an object of the boost::filesystem::path class [1] is declared and will store the path where the code is generated. Listing 6.1 shows this class which will contain in addition all the functions for the code generation.

6.2 Find ports and operations

The first method to declare in this new class has to look into the graph and obtain the inputs and outputs since they will be member variables of the class declared in the generated code. In Listing 6.2, the function parser::in_out_ports() is called to obtain the SymbolCollection for both inputs and outputs. Then the name of each SymbolCollection is stored in a vector of strings and will be used in the code generation.
The order of ports initialization does not influence the result but it is not the same for the operations. When an operator vertex is being analyzed, its operands must have been analyzed before. Therefore, a topological sort on the CDFG is performed to order the vertices.

Topological sort [3] is a way of ordering vertices such that for each edge of the graph, the source vertex appears before the target vertex. In the BGL [2] it is done calling to the `topological_sort()` function which creates a linear ordering of the vertices but in the inverse way.

Before ordering the vertices, it is necessary to filter the graph and remove the cycle assign edges as it was explained before. Then, the reverse topological sort is done to the filtered graph. The parameters of the method are the proper graph, a back inserter iterator which will insert the sorted vertices at the end of the `sorted_vertices` vector and a map to keep track of the ordering process through the graph.
After doing the topological sort, the vector with the vertices is traverse in a reverse way to have the correct order. For each vertex, its type is checked and different methods are called depending on the type. This process can be seen in Listing 6.3.

```
std::string SimulatorCDFG::find_operations(const cdfg_t& g) {
    std::stringstream ostr;
    operations_co_m.clear();
    operations_cy_m.clear();
    operations_condition_m.clear();
    typedef boost::filtered_graph<cdfg_t, filtered_edges> filtered_graph_t;
    filtered_graph_t fg(g, filtered_edges(g));
    vertex_list_t sorted_vertices;
    utils::vertex_index_map_factory<cdfg_t> vertex_index_map(g);
    boost::topological_sort(fg, std::back_inserter(sorted_vertices),
        boost::vertex_index_map(vertex_index_map.get()));
    for (vertex_list_t::reverse_iterator ri = sorted_vertices.rbegin();
        ri != sorted_vertices.rend(); ++ri) {
        const node_info_t& ninf = g[*ri];
        const std::type_info& ti = typeid(*ninf);
        if (ti == typeid(vertex::CombAssign)) {
            operations_co_m.push_back(*ri);
            ostr << generate_assignments_co(g, *ri);
        } else if (ti == typeid(vertex::CycleAssign)) {
            operations_cy_m.push_back(*ri);
            ostr << generate_assignments_cy(g, *ri, operations_cy_m.size());
        } else if (ti == typeid(vertex::Condition)) {
            operations_condition_m.push_back(*ri);
            ostr << generate_conditions(g, *ri);
        } else if (ti == typeid(vertex::BlockStart)) {
            if (name.substr(start+5) == "Switch") {
                ostr << "switch(";
            }
        } else if (ti == typeid(vertex::BlockEnd)) {
            if (name.substr(start+5) == "Switch") {
                ostr << ")" << std::endl;
            }
        }
    }
    return ostr.str();
}
```

Listing 6.3: Method to find operations

If a vertex is a combinational assign vertex, cycle assign vertex or condition vertex, the vertex is stored in a vector and it is called to the proper method that will be
explained below. If it is a BlockStart vertex or a BlockEnd vertex of a switch statement, it only has to open or close the brackets of the statement.

Due to not having more time to continue with this task, the only condition studied is the switch statement. In future work, it is necessary to develop more cases as well as the if/else statement and other conditional statements.

6.2.1 Generating combinational assignments

A combinational assignment in a CDFG can be just an assignment between two ports but also it can come from a conditional statement like a switch. Therefore, the in-coming edges of the assignment vertex are the first thing to check. If one of them is a CaseCondition edge, then it is necessary to write the statement case value: before processing the assignment. The value of the case statement is extracted from the name of the in-coming edge.

The next step is to print the proper assignment which gives a value to its left operand based on the value of its right operand. Checking the out-going edges of the vertex, the one which is a CombAssign edge, its target vertex is the left operand. It can happen that this port is not an input or output, it can be a signal. If it is, it has to be initialize before writing the assignment.

Once the left operand is written, then it is the turn to the symbol equals and the right operand. It can just be an in-coming edge but it can also be the result of operations between in-coming edges. Therefore, the right operand needs to be extracted from the name of the assignment vertex.

Finally, if there was a CaseCondition in-coming edge, it is necessary to print break; to finish the case statement.

Listing 6.4 shows the process followed each time a combinational assign vertex is found.

6.2.2 Generating cycle assignments

The way of generating cycle assignments is quite similar to the combinational assignments except for the need of registers.

In the generated code, there will be two vectors called write_reg and read_reg. The size of the vectors will be the number of cycle assignments in the graph. The write_reg vector is in charge of storing the right operand of the assignment, it simulates the entrance of the register. The read_reg vector refers to the output of the register and its value is assigned to the left operand of the cycle assignment. The way of simulating a clock cycle is with the rising_edge() method where the values of the elements in both vectors are swapped. After the first call to this function, the read_reg vector will have the value of the previous code write_reg vector. If there is only a cycle assignment, it is only necessary one clock cycle and then the left operand will have the value of the right operand. However, if there are more cycle assignments linked or the value of the right operand is changing, it is necessary to simulate more clock cycles.
listing 6.4: method to generate combinational assignments

```cpp
std::string SimulatorCDFG::generate_assignments_co(const cdfg_t& g, const vdesc_t& v) {
    std::stringstream ostr;
    auto in = boost::in_edges(v, g);
    for (auto it = in; it.first != it.second; ++it.first) {
        const edge_info_t& edge_info = g[*it.first];
        const std::type_info& ti = typeid(*edge_info);
        if (ti == typeid(edge::CaseCondition)) {
            edge::Edge& edge = *g[*it.first];
            std::string edge_name = edge.text();
            auto start_edge = edge_name.find("+");
            ostr << "case " << edge_name.substr(start_edge + 1, edge_name.size() - start_edge - 1)
                 << ":" << std::endl;
        }
    }
    auto out = boost::out_edges(v, g);
    for (auto it = out; it.first != it.second; ++it.first) {
        const edge_info_t& edge_info = g[*it.first];
        const std::type_info& ti = typeid(*edge_info);
        if (ti == typeid(edge::CombAssign)) {
            if (is_new_signal(*g[boost::target(*it.first, g)])) {
                ostr << "int ";
            }
            ostr << *g[boost::target(*it.first, g)] << " = ";
            vertex::Vertex& vertex = *g[v];
            std::string name = vertex.text();
            auto start = name.find("'d");
            if (start != std::string::npos) {
                ostr << name.substr(start + 2, name.size() - start - 4) << ";" << std::endl;
            } else {
                auto start_second = name.find("=");
                ostr << name.substr(start_second + 1, name.size() - start_second - 2)
                     << ";" << std::endl;
            }
        }
    }
    for (auto it = in; it.first != it.second; ++it.first) {
        const edge_info_t& edge_info = g[*it.first];
        const std::type_info& ti = typeid(*edge_info);
        if (ti == typeid(edge::CaseCondition)) {
            ostr << "break;" << std::endl;
        }
    }
    return ostr.str();
}
```
In Listing 6.5 is shown the parts referred to generate these assignments with the use of registers.

**Listing 6.5: Method to generate cycle assignments**

```cpp
std::string SimulatorCDFG::generate_assignments_cy(const cdfg_t& g, const vdesc_t& v, 
int size) 
{
    std::stringstream ostr;
    ...
    vertex::Vertex& vertex = *g[v];
    std::string name = vertex.text();
    auto start = name.find("\"d\") - ";
    if (start != std::string::npos) 
    {
        ostr << name.substr(start+2,name.size()-start-2) << ";" << std::endl;
    } 
    else 
    {
        auto start_second = name.find("\"=");
        ostr << name.substr(start_second+1,name.size()-start_second-2) 
        << ";" << std::endl;
    }
    auto out = boost::out_edges(v,g);
    for (auto it = out; it.first != it.second; ++it.first) 
    {
        const edge_info_t& edge_info = g[*it.first];
        const std::type_info& ti = typeid(*edge_info);
        if (ti == typeid(edge::CycleAssign)) 
        {
            ostr << *g[boost::target(*it.first,g)] << " = read_reg[" 
            << size-1 <<"]";
            ostr << std::endl;
        }
    }
    ...
    return ostr.str();
}
```

### 6.2.3 Generating switch statements

As it was mentioned before, the only condition statement simulated is the switch statement. Therefore, it is only necessary to print the variable that will be later compared to the different values of the case statements. This variable is extracted from the name of the vertex.

### 6.3 Generate C++ code

Firstly, a folder with the name of the PU is created in the directory which it is being run. Both generated files `.h` and `.cpp` will be located in this folder.

The way of writing in these files is using the `boost::filesystem::ofstream` class [1] which is a stream class to write on files and indicating the path and the mode `std::ios::out` which means open for output operations.

Listing 6.6 shows how to generate the `.cpp` file. The same process is done for the `.hh` file.
6.3 Generate C++ code

Listing 6.6: Function to generate C++ code

```cpp
void SimulatorCDFG::generate_files(parser::ParseUnit& pu, const cdfg_t& g) {
    std::string cpp_name = boost::lexical_cast<std::string>(pu.name()) + ".cpp";
    std::string name_upper = boost::lexical_cast<std::string>(pu.name());
    path_m = name_upper;
    std::transform(name_upper.begin(), name_upper.end(), name_upper.begin(), toupper);
    utils::open_nogap_dir("test");
    boost::filesystem::path cpp_class(path_m/cpp_name);
    boost::filesystem::ofstream code_cpp(cpp_class, std::ios::out);

    code_cpp << "#include "" << hh_name << "]" << std::endl
    << "#include <iostream>" << std::endl << std::endl
    << "void " << name_upper << "::operator()()" << std::endl
    << "{" << std::endl
    << find_operations(g)
    << "}" << std::endl << std::endl
    << "void " << name_upper << "::rising_edge()" << std::endl
    << "{" << std::endl
    << "std::swap(read_reg,write_reg);" << std::endl
    << "}" << std::endl;
...}
```

6.3.1 Example of code generated

All the process of code generation will be more clear with an example.

The first step is to define an FU with some inputs, outputs and operations. It is written in NoGap CL and it is shown in Listing 6.7. There is a cycle and a comb block with a switch statement included so all the previous functions will be used.

After running NoGap, the CDFG is built for this hardware description and it can be seen in Figure 6.1.

Then, the next step is to generate the C++ code from the graph. The result can be seen in Listing 6.8 and Listing 6.9.

The test.hh file defines the inputs and outputs of the graph in addition to both vectors of size two used as registers. The size is due to there are two cycle assignments. The do_cycle() function handles the call to the functions of the class TEST. After initializing the inputs, it calls to the operator() method, the rising_edge() method and the operator() method again. This is the way of simulating a clock edge.

If there are not cycle assignments and the inputs do not change, in the first clock cycle the outputs will have their final value. However, if the inputs are changing the do_cycle() function should be called to update the output and if there are one or more cycle assignments the same function should be called several times until the outputs have their final value.

The test.cpp file describes the methods of the class TEST. In the operator() method all the operations in the graph are done and the rising_edge() method only swaps the values of the registers.
Listing 6.7: FU specification

```cpp
fu test
{
  input a_i;
  input b_i;
  input c_i;
  output res_o;
  output res2_o;
  signal s_cycle;

cycle
{
  s_cycle = a_i & b_i;
  res_o = s_cycle | c_i;
}

comb
{
  switch (c_i)
  {
    0:%NOP
    {
      res2_o = 1;
    }
    1:%WRITE
    {
      res2_o = 0;
    }
  }
}
```
Figure 6.1: CDFG
#ifndef TEST_HH
#define TEST_HH
#include <iostream>
#include <vector>

class TEST {
  int a_i;
  int b_i;
  int c_i;
  int res2_o;
  int res_o;
  typedef std::vector<int> vec;
  vec read_reg = vec(2);
  vec write_reg = vec(2);
  public:
  void write_input_a_i(const int& val) {a_i = val;}
  void write_input_b_i(const int& val) {b_i = val;}
  void write_input_c_i(const int& val) {c_i = val;}
  const int& read_output_res2_o() const {return res2_o;}
  const int& read_output_res_o() const {return res_o;}
  void operator()();
  void rising_edge();
};

void do_cycle(TEST& t) {
  t.write_input_a_i(1);
  t.write_input_b_i(1);
  t.write_input_c_i(1);
  t.operator()();
  std::cout << "res2_o = " << t.read_output_res2_o() << std::endl;
  std::cout << "res_o = " << t.read_output_res_o() << std::endl;
  t.rising_edge();
  t.operator()();
  std::cout << "res2_o = " << t.read_output_res2_o() << std::endl;
  std::cout << "res_o = " << t.read_output_res_o() << std::endl;
};
#endif
# Listing 6.9: test.cpp

```cpp
#include "test.hh"
#include <iostream>

void TEST::operator()() {
    int s_cycle = 0;
    write_reg[0] = (s_cycle|c_i);
    res_o = read_reg[0];
    write_reg[1] = (a_i&b_i);
    s_cycle = read_reg[1];
    switch(c_i) {
        case 1:
            res2_o = 0;
            break;
        case 0:
            res2_o = 1;
            break;
    }
}

void TEST::rising_edge() {
    std::swap(read_reg,write_reg);
}
```

6.3 Generate C++ code
Once the CDFG has fulfilled the requirements, some information is extracted from the graph in order to be used in future work. This refers to obtain all possible paths from an input to an output. Then, it is possible to check the edges of the paths and know if there is any cycle assignment or not. If there is, it means that there would be a delay in the path.

### 7.1 Search for all paths

BFS [3] is a traversal through a graph that explores all of the vertices reachable from a particular source vertex. The difference with DFS is the order of the traversal; the algorithm explores all the neighbors of a vertex before going deeper. This process creates a breadth-first tree for each source vertex, so it will be necessary to run the algorithm from different source vertices in order to find all the possible paths.

The `breadth_first_search()` function [2] performs a breadth-first traversal of a directed or undirected graph. Its parameters are a color map and a queue to implement the traversal together with the graph and a source vertex. This function can be extended through a visitor the same way it was explained for DFS. In Listing 7.1 can be seen how the search is done for each input.

The structure that is used as a visitor, see Listing 7.2, inherits from `bfs_visitor<>` which provides the visitor with empty versions of the BFSVisitor methods. In this case, the only method needed to override is the `examine_edge()` which is invoked on every out-edge of each vertex after it is discovered. The edge is stored in a vector and will be used to print the different paths.
Listing 7.1: Use of BFS

```cpp
for (const auto& in: inputs)
{
    boost::queue<vdesc_t> q;
    utils::vertex_index_map_factory<cdfg_t> vertex_index_map(graph);
    utils::color_map_factory<cdfg_t> color_map(graph, vertex_index_map);
    std::vector<edesc_t> edges;
    record_paths vis(edges);
    boost::breadth_first_search(graph, in, q, vis, color_map.get());
    ...
}
```

Listing 7.2: Visitor for BFS

```cpp
struct record_paths : public boost::bfs_visitor{
    protected:
    std::vector<edesc_t>& edges_m;
    public:
    record_paths(std::vector<edesc_t>& edges_) : edges_m(edges_) {}
    template <class Edge, class Graph>
    void examine_edge(Edge e, Graph&) const
    {
        edges_m.push_back(e);
    }
};
```

An alternative to the previous solution could have been the use of a predecessor recording visitor [2]. For each examined edge, the source vertex would be recorded as the predecessor of the target vertex. The edges would be also needed due to it is necessary to check if there are cycle assign edges or not. Then all the possible paths could be accessed through the predecessor map created.

However, this has not been the chosen solution, it has been the one explained before which operates similar to the use of the visitor in the DFS: only the edges are stored and after they can be printed with the `show_path()` method that will be explained below. Apart from being easier due to the similar operations, the alternative solution has also the disadvantage of handling the access to the map.

### 7.2 Show all paths

The next step is to print the different paths and check if there will be a delay which means that there is a cycle assign edge. Just after calling the `breadth_first_search()` function for each input, for each output it is called the `show_path()` function as it is shown in Listing 7.3.

This method needs a vector of edges and a queue of these vectors for printing all paths. The input vertex is the first one to handle with. The first step is to check all the edges obtained with the `breadth_first_search()` if its source vertex is the input vertex. This means searching all the out-going edges from the input vertex.
Listing 7.3: Call to the show_path method

```cpp
std::stringstream ss;

for (const auto &out : outputs)
{
    ss << "From " << *graph[in] << " to " << *graph[out];
    ss << show_path(edges, graph, in, out);
}
```

vertex. If the edge has not been stored in the vector yet, a new vector is created as a copy of the previous one and this last edge is pushed to the new vector. Then the new vector is pushed to the queue of vectors.

Once there is at least one element in the queue, the same process will be done until the queue is empty. Firstly, it consists on extracting the first element of the queue (the oldest one). Now the vertex to handle with is the target of the last edge in the vector popped out. If the vertex is the output of the path, the process has finished. But if not, each out-going edge of the vertex is stored in a new vector built as a copy of the previous one and the new vector is pushed to the queue as it was done with the input vertex.

This process continues until the queue is empty which means that there are not more paths to be studied. Each time the output vertex is reached, it is checked if there is an edge between the output and the input. If there is, it means that the path is a cycle and the function has been called from Listing 4.3. The edges passed as a parameter in this case are the ones obtained from the depth_first_search(). When a cycle is found, it is printed the source and target vertices of all the edges in the path and the type of each edge.

On the other hand, if there is not a cycle, then the source and target vertices from each edge are printed and it is checked if it is a cycle assign edge. If it is, the path will have a delay and a register is needed, the delay variable will have a non-zero value. This way all the paths between inputs and outputs are printed together with delay which can take the value 1 (delay) or 0 (no delay).

The code of this process is shown in Listing 7.4.
Listing 7.4: Method to show all paths

```cpp
std::string ProcessControlDataFlowGraph::show_path(const std::vector<edesc_t>& edges, const cdfg_t& g, const vdesc_t& input, const vdesc_t& output) const
{
    std::stringstream ss;
    std::vector<edesc_t> path;
    std::queue<std::vector<edesc_t>> q;
    auto last_vertex = input;
    for(const auto& index : edges)
    {
        if(source(index, g) == last_vertex)
        {
            if(edge_not_in_current_path(index, path))
            {
                std::vector<edesc_t> new_path(path.begin(), path.end());
                new_path.push_back(index);
                q.push(new_path);
            }
        }
    }
    while(!q.empty())
    {
        path = q.front();
        q.pop();
        last_vertex = boost::target(path[path.size()-1], g);
        if(last_vertex == output)
        {
            if(boost::edge(output, input, g).second)
            {
                path.push_back(boost::edge(output, input, g).first);
                for(const auto& index : path)
                {
                    std::type_info* edge_info = g[index];
                    const std::type_info* ti = typeid(*edge_info);
                    ss << "( " << *g[boost::source(index, g)] << " , " << *g[boost::target(index, g)] << " ) \n";
                    edge_info_t edge_info = g[index];
                    const std::type_info* ti = typeid(*edge_info);
                    ss << ti->name();
                }
            }
            else
            {
                int delay = 0;
                for(const auto& index : path)
                {
                    ss << "( " << *g[boost::source(index, g)] << " , " << *g[boost::target(index, g)] << " ) \n";
                    edge_info_t edge_info = g[index];
                    const std::type_info* ti = typeid(*edge_info);
                    if(ti == typeid(edge::CycleAssign)) delay = 1;
                }
                if(delay) ss << " \nDelay = 1 \n";
                else ss << " \nDelay = 0 \n";
            }
        }
        else
        {
            if(source(index, g) == last_vertex)
            {
                if(edge_not_in_current_path(index, path))
                {
                    std::vector<edesc_t> new_path(path.begin(), path.end());
                    new_path.push_back(index);
                    q.push(new_path);
                }
            }
        }
    }
    return ss.str();
}```
Conclusions and future work

The aim of this thesis was to validate the CDFGs generated by NoGap. The way to accomplish it was adding functionality to NoGap through the program explained in this report. From different FU descriptions, their CDFGs were generated. After that, the graphs were checked to fulfill the requirements. The result was that all the CDFGs have no cycles, their inputs and outputs have only out-edges and in-edges respectively and it is possible to sequentialize the graphs generating C++ code.

However, this last requirement has not been fully accomplished. In the generation of the C++ code, the switch statements are the only condition vertices that are analyzed. It is necessary to analyze and write the proper C++ code for the if/else statements too.

Another aspect to improve is the use of substring to generate the C++ code. For example in a case statement, the different values of the condition are read from the name of the edge using substring. It also happens in an assignment when the right operand is an operation between different vertices. Then, it is necessary to use substring to know the relation between the in-edges of the assignment. This fact is error prone and has to be changed.

In the generated code if there is more than one cycle assignment and they are linked, the order of the generated instructions is not the right one. Due to the removal of the cycle assignments, when the topological sort is done the first cycle assign vertex to analyze is the deepest one in the graph. And this is an error, that one should be the last one. Therefore, the generated code in Listing 6.9 is not totally precise even though the results were correct. Lines 9 and 10 should appear before lines 7 and 8.
A way to solve this problem is not removing the cycle assignments at first. The cycle assign vertex has to be split into two, one for the read side and the other one for the write side. These nodes will be differentiated by a number that can be the same one used as the index in the register vector. Then, the topological sort is done and the nodes are analyzed in the right order. The cycle assign vertices can be treated as combinational vertices. This way, the generated code will reflect how the hardware works.

Moreover, some information has been extracted from the graph. All possible paths between the inputs and outputs are studied and checked if there would be a delay in the path or not. Therefore, the user does not need to add a delay to the block when operating in the high level graph, the delay is automatically checked.
[1] Boost filesystem library

[2] Boost graph library


[7] C++11
**Upphovsrätt**

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida [http://www.ep.liu.se/](http://www.ep.liu.se/)

---

**Copyright**

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: [http://www.ep.liu.se/](http://www.ep.liu.se/)

© Mónica Sánchez Yagüe