GML-DFG: An Implementation of Dependence Flow Graphs in Graph Modeling Language

Ntšibane Ntatlapa and Richard Chapman

Abstract

We present an implementation of dependence flow graphs (DFGs) in graph modeling language (GML). This implementation will be used as an intermediate form for a high-level synthesis system. We describe the attributes of edges and nodes in GML-DFG and then present their syntax in terms of the key-value pairs that describe the attributes. We also describe the subgraphs that can be used to represent control constructs.

Computer Science and Engineering,
Auburn University,
Auburn, Alabama
March 30, 1999
# Contents

1 Introduction .................................................. 3

2 Graph .......................................................... 6

3 Edges ............................................................ 7

4 Nodes ............................................................ 8
  4.1 Operator nodes ............................................. 10
  4.2 I/O operator nodes ........................................ 10
  4.3 Control Nodes .............................................. 11
    4.3.1 Switch and Merge ..................................... 11
    4.3.2 Loop and exit ......................................... 11
  4.4 Wait node .................................................. 12
  4.5 Other nodes ................................................ 12
    4.5.1 Constant node ......................................... 12
    4.5.2 start and end nodes .................................. 13
    4.5.3 Param and return nodes ............................... 13
    4.5.4 Put and Get nodes .................................... 13
    4.5.5 Synchronization nodes ............................... 14

5 Example of a Simple Graph .................................... 14

6 Graph with Aggregates ........................................ 15
  6.1 If aggregate ................................................ 15

7 Loop Aggregate ................................................ 16

8 Call Aggregate ................................................ 17

9 Wait Aggregate ................................................ 17

10 Conclusion ................................................... 18

## List of Figures

1 Dfg for straight-line code ..................................... 4

2 Dfg with resource dependences .............................. 4

3 Dfg with control dependences ............................... 5

4 Example of a DFG with wait statement .................... 6

5 Wait Aggregate ................................................ 18
List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Mandatory Graph Attributes</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Optional Graph Attributes</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Mandatory Edge Attributes</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Optional Edge Attributes</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Mandatory Node attributes</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Optional Node attributes</td>
<td>10</td>
</tr>
</tbody>
</table>
GML-DFG: An Implementation of Dependence Flow Graphs in Graph Modeling Language

Ntšibane Ntlatlapa
ntlatnse@eng.auburn.edu

Richard Chapman
chapman@eng.auburn.edu

March 30, 1999

Abstract

We present an implementation of dependence flow graphs (DFGs) in graph modeling language (GML). This implementation will be used as an intermediate form for a high-level synthesis system. We describe the attributes of edges and nodes in GML-DFG and then present their syntax in terms of the key-value pairs that describe the attributes. We also describe the subgraphs that can be used to represent control constructs.

1 Introduction

We are working on the development of a high-level synthesis system. This system starts with a user-specified behavioral description of the design written in behavioral VHDL. The description is transformed into an internal format, dependence flow graph, which is subsequently used for scheduling and data path allocation. This results in a chip layout written in structural VHDL.

Dependence flow graphs [PBJ91] model the data, resource, and control dependences present in a program. Consider the pseudo-code fragment in Figure 1. The associated graph has as its nodes the operations in the program, and as its edges the data dependences between the operations. Consider the graph as a model for execution of the operations in the program. The incoming edges to a node express all the data dependences for that node. All nodes at the tails of incoming edges to a node must execute before that node can execute. In fact, for this simple program with no resource or control dependences, this is the dependence flow graph for the program.

Other sorts of dependences exist in programs specifying hardware. Consider a program specifying a circuit with two ports: an input port A and a bidirectional port B. The program in Figure 2 specifies part of the behavior of such a circuit. The two operations that access port A and the three that access port B must be ordered somehow, to guarantee that the operations will not ever be in contention for access to a port. These dependences resemble the dependences in software programs between accesses to the same memory location. However, in addition to the read-after-write, write-after-read, and
write-after-write dependences in software, we also have read-after-read dependences on input operations, since the value present at an input port is set by the external world. Handling I/O dependences is the one extension we made to the graph formalism itself to describe hardware [CBL92].

The graph in figure 2 represents the data and resource dependences present in the program in that figure. The resource dependence arrows indicate a partial ordering on the scheduling of I/O operations. Any valid execution schedule for the DFG must respect both the orderings imposed by the data and resource dependences.

![Diagram](https://example.com/diagram.png)

**Figure 1:** Dig for straight-line code

![Diagram](https://example.com/diagram.png)

**Figure 2:** Dig with resource dependences
In addition to statements describing computation and I/O behavior, HDLs provide constructs to allow the conditional execution of a part of a program, or to repeat execution of a part of a program as long as some condition holds. This information must be included in the intermediate form if it is to be an adequate representation of the program for scheduling and datapath allocation. Dependence flow graphs model these control dependences by routing the data and resource dependences based on control information. Consider the first program and graph in Figure 3. The switch nodes route the values of a and b to the add if the condition \( x=0 \) holds, else they route those values to the subtract; only one arithmetic node or the other will execute. The merge node’s behavior is to execute whenever exactly one of its inputs takes a value, and to propagate that value to the output edge.

Switch and merge nodes can also be used to represent loops in dependence flow graphs, as in the second program and graph in Figure 3. In the future we will refer to the merge node at the entry to a loop as a “loop” node and the switch at the exit from a loop as an “exit” node.

![Diagram](image)

(a) Selective Execution  
(b) Repeated Execution

Figure 3: DFG with control dependences

Programs written in HDLs may specify timing dependences. In VHDL \[1987\] the wait construct can be used to specify this kind of dependence. Timing dependences are similar to read-after-read dependences on input operations. Figure 4 is an example of a DFG program with wait nodes.

Graph modeling language \[Him96, Him97\] is a portable file format for graphs. A GML file consists of a hierarchical key-value lists. Graphs can be annotated with annotated with arbitrary data structures. We are adapting this format for use in our high-level synthesis project.

The rest of the document describes the key-value pairs that can be used in GML files describing dependence flow graphs. We will use the keys, graphs, node, edge, id, source and target as defined in \[Him96\]. The rest of the keys that we describe here together with the standard ones above will be the ones that our system can process; there may be others which will be treated as comments.
2 Graph

A graph in GML-DFG is described by a key “graph”. The value for this key is a list of nodes and edges describing the graph operations and communications between them. These are mandatory attributes of a graph. Optional attributes provide extra information on the graph. Users are free to add their own attributes. These attributes will be treated as comments. Tables 1 and 2 list mandatory and optional attributes of a graph respectively.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>List</td>
<td>Nodes describe operations in a graph. A valid graph must have at least two nodes.</td>
</tr>
<tr>
<td>edge</td>
<td>List</td>
<td>Edge communicating results between graph operations.</td>
</tr>
</tbody>
</table>

Table 1: Mandatory Graph Attributes
<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Identity of the graph. Must be unique for all graphs in a file.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the graph.</td>
</tr>
<tr>
<td>kind</td>
<td>String</td>
<td>A construct that this graph represent. Legal values include procedure, function and architecture.</td>
</tr>
<tr>
<td>comment</td>
<td>String</td>
<td>A subgraph (aggregate) that represent a conditional statement.</td>
</tr>
<tr>
<td>if</td>
<td>List</td>
<td>A subgraph (aggregate) that a represent a subprogram call.</td>
</tr>
<tr>
<td>call</td>
<td>List</td>
<td>A subgraph (aggregate) that represent repeated execution.</td>
</tr>
<tr>
<td>while</td>
<td>List</td>
<td>A subgraph (aggregate) that represent wait statement execution.</td>
</tr>
</tbody>
</table>

Table 2: Optional Graph Attributes

3 Edges

To describe the edge we use the key “edge”. The value is the list of attributes describing the edge. As in the case of a graph we classify the edge attributes as mandatory and optional. Mandatory attributes are described in Table 3 and optional attributes in Table 4.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>An identity of an edge. Must be unique for all edges in a graph.</td>
</tr>
<tr>
<td>source</td>
<td>Integer</td>
<td>An identity of a source node.</td>
</tr>
<tr>
<td>srcport</td>
<td>Integer</td>
<td>An identity of an output port of a source node that an edge is connected to. More than one edge can be connected to the same output port.</td>
</tr>
<tr>
<td>target</td>
<td>Integer</td>
<td>An identity of a target node.</td>
</tr>
<tr>
<td>trgport</td>
<td>Integer</td>
<td>An identity of an input port of a target node. Only one edge can be connected to the input edge.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Type of a value carried by the edge. Generally edges carry values produced by the source nodes, however some edges carry dependence tokens. Therefore, the legal values for a type are all the types that are defined for nodes plus a special dependence type.</td>
</tr>
</tbody>
</table>

Table 3: Mandatory Edge Attributes
<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>String</td>
<td>The variable or an I/O resource from the program that created the graph. This will be useful for debugging.</td>
</tr>
<tr>
<td>comment</td>
<td>String</td>
<td>Defines a comment. Used for documentation.</td>
</tr>
</tbody>
</table>

Table 4: Optional Edge Attributes

Each edge has exactly one source and one target. There may be more than one edge with the same source and the same source port. For example, the value of “y” in Figure 1 is carried by two edges. Both of them are connected to the same output port of a read node. Only one edge can be connected to an input port of a target node. The following program segment shows the definitions of the edges that carry the value of in Figure 1.

```plaintext
1  graph [....
2    node [  
3      id 1  
4      op "read"  
5      ....  
6    ]  
7    edge [  
8      id 1  
9      source 1  
10     srcport 2  
11     ....  
12    ]  
13    edge [  
14      id 2  
15      source 1  
16     srcport 2  
17     ....  
18    ]  
19    ....  
20 ]
```

4 Nodes

To describe the GML-DFG nodes we use the key “node”, and its value is a list of mandatory and optional attributes of a node. Unlike edges, there are many different kinds of nodes, the most general ones being: functional operator nodes, I/O operator nodes and control nodes. We will describe the mandatory attributes and optional attributes for nodes, however some of the optional attributes may be mandatory for some kinds of nodes. Tables 5 and 6 describe mandatory and optional attributes respectively.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>An identity of an node. Must be unique for all nodes in a graph.</td>
</tr>
</tbody>
</table>

P.T.O....
<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| op   | String | An operation performed by the node.  
      |        | - add, sub, mult, div, and, or, xor, nand,  
      |        |   nor, mod, rem, eq, neq, lt, leq, gt, geq,  
      |        |   abs, not, concat.  
      |        | - switch, merge, loop, exit  
      |        | - wait  
      |        | - sync0r, syncAnd, start, end, param,  
      |        |   return, put, get, read, write. |
| type | String | The type of a node. Any string defined as a type can be  
      |        | a valid entry. The standard types are integer, float,  
      |        |   boolean, bit and bit-vector. |

Table 5: Mandatory Node attributes

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment</td>
<td>String</td>
<td>Defines a comment. Used for documentation.</td>
</tr>
<tr>
<td>srcline</td>
<td>Integer</td>
<td>The line number of the statement that generated the node. Used for debugging.</td>
</tr>
<tr>
<td>imports</td>
<td>List</td>
<td>List of input port ids. For most of the nodes the operation of the node dictates the number of input ports, e.g. add node has two inputs. Some nodes may have variable number of input ports. The value of this key is list that has key-value pairs giving the identities of the input ports. This is useful for documentation purposes only for most of the nodes.</td>
</tr>
<tr>
<td>outport</td>
<td>List</td>
<td>List of output port ids. The number of outputs is predetermined by the operation performed by the node. This attribute is optional because some nodes do not have output ports. The value of this key is a list that has key-value pairs giving the identities of the output ports. This is useful for documentation purposes only since the output ports are predetermined by the operation.</td>
</tr>
<tr>
<td>value</td>
<td>Literal</td>
<td>The value assigned to the node. Useful for constant nodes only. The value for the 'value' key should be a constant literal of the node's type.</td>
</tr>
<tr>
<td>leftbit</td>
<td>Integer</td>
<td>The left bit in a select node. Does not mean anything for all other nodes.</td>
</tr>
</tbody>
</table>

continue....
<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rightbit</td>
<td>Integer</td>
<td>The right bit in a select node. Does not mean anything for all other nodes.</td>
</tr>
<tr>
<td>port</td>
<td>String</td>
<td>Defined for I/O operations read and write only. This is the port on which an operation is performed.</td>
</tr>
<tr>
<td>timeout</td>
<td>String</td>
<td>Defined for a wait node only. It defines the time the wait should suspend for.</td>
</tr>
<tr>
<td>assoc</td>
<td>Integer</td>
<td>Defined for put and get nodes. It associates each put or get node with param and return nodes of another graph.</td>
</tr>
</tbody>
</table>

Table 6: Optional Node attributes

The optional attributes comment, srcline and port are defined for all nodes. The attributes srcline and port are used for debugging and comment is used for documentation purposes only. The other optional attributes leftbit, rightbit, value and timeout are defined for some of the nodes.

4.1 Operator nodes

For the operator nodes, the value for the key “op” is the functional operator that is applied to the inputs. Operator node must have a type attribute. The values for imports and outputs keys are preset as follows: opLEFT is set to 1, opRIGHT is set to 2 and opOUT is set to 1.

Unary operator nodes will ignore right operand. The following GML-DFG nodes are examples of binary operator nodes and unary operator nodes respectively.

```
1       node Lid   1
2       type      "integer"
3       op        "add"
4       comment   "a + b"
5       sourceline 5

6       ]

7       node Lid   2
8       type      "int"
9       op        "not"
10      comment   "not b"
11      sourceline 4
12      ]
```

4.2 I/O operator nodes

These nodes must have the attributes type and port. The type of the node corresponds to the type of a port that the operation is on. The values of imports and outputs for a read node are preset as follows: readIN is set to 1, readOUT is set to 1 and readVAL is set to 2. The write node has the following presets for imports and outputs: writeIN is set to 1, writeVAL is set to 2 and writeOUT to 1.

Below are examples of GML-DFG read and write nodes respectively.
4.3 Control Nodes

4.3.1 Switch and Merge

The value of the attribute op for the GML switch nodes is “switch” and for merge it is “merge”. Both switch and merge nodes must have types and the types of incoming edges (except for the edge connected to swPRED which must be of boolean type) and outgoing edges must match the node type. Switch node has imports attribute with the following predefined values: swPRED is set to id 1 and swIN is set to 2. The output attribute is preset as follows: swTRUE is set to 1 and swFALSE is set to 2. The GML node below is an example of a switch node similar to the one in Figure 3(a).

```
1 node [id 1
2     type "integer"
3     op "switch"
4     comment "if a > 4"
5     srcline 5
6 ]
```

The imports for merge node are preset to 1 for mgLEFT and 2 for mgRIGHT. The merge has one output mgOUT and it is preset to 1. The GML below is an example of a GML-DFG merge node.

```
1 node [id 1
2     type "integer"
3     op "merge"
4     comment "if a > 4"
5     srcline 5
6 ]
```

The switch and merge nodes must be in an aggregate of kind “If”, see section 6.

4.3.2 Loop and exit

Loop and exit nodes are similar to merge and switch nodes respectively. They must have an attribute type. The values of imports and outputs are preset as
follows: For loop nodes, loopIN is set to 1 and loopBACK is set to 2. The output port loopOUT is set to 1. For exit nodes, exitPRED is set to 1 and exitIN is set to 2. The output ports are preset as follows: exitTRUE is set to 1 and exitFALSE is set to 2. The following are examples of loop and exit nodes:

```
node [id 1
  type "integer"
  op "exit"
  comment "while a > 4 loop"
]

node [id 1
  type "integer"
  op "loop"
  comment "while a > 4 loop"
]
```

The loop and exit nodes must appear in an aggregate of kind "loop".

### 4.4 Wait node

The wait node has the input ports preset as follows: waitIN is set to 1, waitSENS is set to 2, waitBACK is set to 3. The output ports are preset as follows: waitTIME is set to 1 and waitOUT is set to 2. This is an example of a wait node:

```
node [id 1
  op "wait"
  timeout "2 ns"
  comment "wait on a for 2 ns"
]
```

### 4.5 Other nodes

#### 4.5.1 Constant node

A constant node has one input and one output. All other nodes have an implicit input edge of type edge, a constant has to have an explicit control input. The input port for a constant constCTRL is preset to 0. Its output port constOUT is preset to 1. A constant node must have type and value attributes. This an example of a constant node:

```
node [id 3
  type "integer"
  op "const"
  value 45
  comment "45"
  sourceline 5
]
```
4.5.2 start and end nodes

These nodes are used to mark the beginning and end of a thread of dependence in a graph. A start node represents locally declared variable or signal. It must have a type attribute. It has only one output and no inputs. Its output startOUT is preset to 1. An end node must also have a type. It has one input and no outputs. Its input port endIN is preset to 1.

```plaintext
define (node [id 1
    op "start"
    type "integer"
  ]

define (node [id 2
    op "end"
    type "integer"
  ]
```

4.5.3 Param and return nodes

Param and return nodes form an interface to the graph. Param nodes get information into the graph while return nodes sends the information out of the graph. A param node has one output and no input. It must have a type attribute. Its output port paramOUT is preset to 1. A return node has one input and no output. Its input port returnIN is preset to 1.

```plaintext
define (node [id 1
    op "param"
    type "integer"
  ]

define (node [id 2
    op "return"
    type "integer"
  ]
```

4.5.4 Put and Get nodes

These nodes make associations between values in the current graph and parameters of another graph. The put and get nodes must correspond to param and return nodes respectively. Both put and get nodes must have types. A put node has one input port putIN which is preset to 1. A get node has one output getOUT which is preset to 1. Both of these nodes have association attribute. It is described by a key-value pair with the key "assoc", the value of this key is an id of a param or return node that it passes the value to or from. These are examples of these nodes:

```plaintext
define (node [id 3
    op "put"
    assoc 1
  ]
```
4.5.5 Synchronization nodes

We provide two synchronization nodes, syncAnd and syncOr. A syncOR node fires as soon as it gets one of its input while syncAND fires when it has all of its inputs. Both these nodes have variable number of inputs, as a result they must have imports attribute since they cannot have preset input ports. The following nodes are examples of syncOr and syncAnd:

```
node [id 1
  op  "syncOr"
  comment "The operation performed on input is an or"
  sourceLine 5
]
```

```
node [id 1
  op  "syncAnd"
  comment "The operation performed on inputs is an and"
  sourceLine 5
]
```

5 Example of a Simple Graph

The simplest DFG is the straight line DFG such as the one in Figure 1. It shows flow dependences in a program. The following is the GML graph for the second DFG in Figure 1.

```
graph [id 1
  comment "A simple DFG graph"
  node [id 1
    type  "integer"
    op  "add"
    comment "z = x + y"
  ]
  node [id 2
    type  "integer"
    op  "read"
    comment "read z"
  ]
  node [id 3
    type  "integer"
    op  "read"
    comment "read x"
  ]
  node [id 4
    type  "integer"
    op  "write"
    comment "write z"
  ]
  edge [id 1
```

14
6 Graph with Aggregates

An aggregate represents a control region or regions, and it is implemented as a subgraph. We currently define three kinds of aggregates: "If" aggregate to represent an if statement, "loop" aggregate, "call" aggregate and "wait" aggregate. The aggregates can be viewed as functions of the form: \( \text{OUT} = \text{fun IN} \), where \( \text{IN} \) is the set of inputs and \( \text{OUT} \) is a set of outputs.

6.1 If aggregate

If aggregates are used to represent conditional statements. The input set for If aggregates is made up of input edges to the switch nodes and the output set is made up of output edge from merge nodes. To describe an if aggregate we use the keywords: "if", "then" and "else". The value for if keyword is a list of switch nodes and merge nodes, and their output edges together with nodes that compute a condition. The value for then is a list of nodes that gets executed if the condition is true if the value for else is a list of nodes that get executed if the condition is false. The GML-DFG subgraph for the DFG in Figure 3(a) is given below.

```
edge [id 6
  source node before switch
  srcPort Port of node before switch
  target 1
  tgtPort 2
  type "integer"
  comment "Edge connecting body A to merge node"
] if {
  node [
    id 1
```
7 Loop Aggregate

Loop aggregates represent repeated executions. A loop aggregate is made up of
loop nodes and exit nodes as well as the nodes that form the body of a loop.
The inputs to the loop aggregate are the input edges to loop nodes. The outputs
are the false or true outputs of exit nodes depending on whether they are values
of while or exit keys. A loop aggregate will be described by key-value pairs with
keys “loop” and “while”. The value of a key while is a list of loop nodes, nodes
that computes condition and exit nodes. The false outputs of these exit nodes
are the outputs of the aggregate. The value of a key loop are the nodes that
form the body of the loop. For example the loop in a program segment below
is represented by the following GML-DFG code:

code...

12 ...compute condition"
13 type "integer"
14 ...
15 edge [id 1
16 op "switch"
17 type "integer"
18 ]
19 edge [id 1
20 source 1
21 srcPort 1
22 target node/s in body A
23 tgtPort Port/s of node/s in body A
24 type integer
25 comment "Edge connecting switch node to body A"
26 ]
27 edge [id 2
28 source 1
29 srcPort 2
30 target node/s in body B
31 tgtPort Port/s of node/s in body B
32 type integer
33 comment "Edge connecting switch node to body B"
34 ]
35 then [Nodes in body A if it is not empty
36 else [Nodes in body B if it is not empty,
37 ...
38 node [id 2
39 op "merge"
40 type "integer"
41 ]
42 edge [id 4
43 source 2
44 srcPort 1
45 type integer
46 comment "Edge connecting switch node to body A"
47 ]
48 ]
49 ]
8 Call Aggregate

Call aggregate represent subprogram calls. The inputs to this aggregate are the input edges of put nodes and outputs are output edges of get nodes. This aggregate is described by a key-value pair with a key “call”, the value of this key is a list of put and get nodes. On top of these nodes, the list includes an association attribute. Association attribute associates a call aggregate to a graph that it calls. The key-value pair describing this attribute and its value is the name of the graph that is being called. For example, a code segment below is a call for a graph named “add” with the following param and return nodes:

```
node [  
id 1  
op param ...]  
node [  
id 2  
op return ...]
```

9 Wait Aggregate

The wait aggregate models the wait statement in VHDL. It suspends the execution until either it times out or the wait condition is satisfied. It also updates the values of the signals. The wait aggregate is made up of wait-exit node pair.

```
while a < 10 loop  
a := a + 1;  
end loop;
```
Figure 5: Wait Aggregate

Figure 5 show the wait aggregate. Unlike in other aggregates such as if aggregate where every signal/port has its own switch node, in wait aggregate we have only wait node and one exit node. A wait aggregate is described by key-value pairs with keys “wait” and “until”. The value of a key “wait” is a wait node and the value of a key “until” is a list of nodes that compute a condition, an or node and an exit node. Before every node aggregate, there must be a syncAND node, the inputs this node are edges that represent all signals that are visible to a VHDL process. The must also be a syncOR node, its inputs are all signals in the sensitivity list of a wait statement. A wait aggregate must be followed by read nodes for all the signals that are accessed in the body of a graph. For example this VHDL wait statement

```vhdl
wait on a, b until a = b for 2 ns;
```

is represented by the following GML-DFG code segment:

```gml
... wait [
  node [...
    op wait
    timeout 2ns ...
  ]
  until [
    node [...
      op eq ...
    node [...
      op or ...
    node [...
      op exit ...
  ]
...]
```

10 conclusion

This is the first draft of the GML-DFG proposed standard. We are currently in the process of building a library to support to this standard. Currently the

18
library has routines to create and print nodes and edges, connect nodes with
graphs. The user of this library should obtain all the include
files in /proj/hls/gml.dfg/include on the engineering network. The user
must also obtain the library file libgml.dfg.a from /proj/hls/gml.dfg/lib

References

in BEDROC. In Proceedings of 1992 European Design Automation


[Him97] Michael Himsmot, GML:a portable graph file format. Technical re-
port, Universität Passau, 1997.

IEEE std 1076, IEEE Press.

[PBJS91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudghill,
and Paul Stodghill. Dependence flow graphs: An algebraic approach
to program dependencies. In 18th Annual ACM Symposium on Prin-