MASTER

Graphical simulation of data flow graphs

Geurts, L.J.F.

Award date:
1993

Link to publication
Graphical simulation of Data Flow Graphs

L.J.F. Geurts

Master Thesis

performed: April 1992 - June 1993
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Ir. J.W.G. Fleurkens

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports
Abstract

Data flow graphs (DFG) are commonly used as a system representation during the silicon compilation process. In the Design Automation Section of the Department of Electrical Engineering at the Eindhoven University of Technology, a DFG system representation has been developed to describe the behaviour of digital hardware. The graph is to be generated from algorithms described in a hardware behaviour description language (HDL) and serves as an intermediate format for circuits during architectural synthesis. In a DFG, nodes represent operations, directed edges represent the inter-relations between nodes. Edges can contain data values, which are modeled by tokens.

ESCAPE is a digital schematic capture program extended with an interactive event-driven simulator. Two views are provided: a symbol view to design the graphical layout of modules and a network view to compose systems of symbols. These symbols either represent primitive modules or composite modules. Each primitive module has a behaviour description specified in a Lisp-like HDL. A simulator is used to evaluate the overall behaviour of the system. Simulation results are presented using animation techniques.

In order to handle different graph models, ESCAPE has to be extended with a graph view for the interactive editing and displaying of graphs. In order to display these models, graphical descriptions of nodes and edges can be specified in a special purpose language. The definition of the DFG and related graphs have been developed to support their usage in ESCAPE. Besides interactive editing, a textual graph format can be used to capture these graphs. Two algorithms are provided for the automatic placement of graphs.

The token flow simulator provides two modes: either a DFG can be simulated stand-alone or in combination with a control graph (CTG). The execution of a node comprises of fetching a token from each incoming edge, using their values to calculate the values of the output tokens, which are passed to the outgoing edges. Two types of tokens are provided: numeric tokens are used to store numbers and symbolic tokens are used to store algebraic expressions. Depending on the input token types, a node is either simulated numeric or symbolic. In case of symbolic simulation, tokens can be simplified by using a term rewrite system.

Possible application areas for the simulator are behaviour verification and in the educational field. A mixed-level mode simulation of graphs with ESCAPE's discrete event
simulator is still under development. Another application could be the simulation of all kinds of HDLs by using the DFG standard as an intermediate format in behaviour simulation. The application area could be enlarged by making the simulator operate on different types of token flow graph models.
# Contents

1 Introduction .......................... 1
   1.1 Data flow graph simulation .................. 2
   1.2 Contents of the report ........................ 2

2 Data flow graphs .......................... 4
   2.1 Position of the DFG format .................. 4
   2.2 DFG semantics .............................. 5
      2.2.1 Operation nodes ......................... 5
      2.2.2 Input and Output nodes ................... 6
      2.2.3 Constant node ........................... 6
      2.2.4 Branch and Merge nodes ..................... 7
      2.2.5 Entry and Exit nodes ....................... 7
      2.2.6 Get and Put nodes ........................ 7
      2.2.7 Array operations ......................... 8
      2.2.8 Other operations .......................... 8
   2.3 DFG examples .............................. 8

3 Extending ESCAPE for DFG simulation .......... 11

4 Graphtype Definition Language ................ 13
4.1 Requirements ......................................................... 13
4.2 Development ......................................................... 15
4.3 Some examples ....................................................... 15

5 The graph view .......................................................... 18
5.1 The graphical structure of ESCAPE ................................. 18
5.2 Extending the graphical structure of ESCAPE for graphs .......... 21
5.3 The network representation of a graph ............................... 22

6 Reading the graph textformat ........................................... 25
6.1 Parsing ASD files ...................................................... 25
  6.1.1 Construction of the parser ...................................... 26
  6.1.2 Extending the DFG parser for ASD files ....................... 28
6.2 Graph placement ...................................................... 30
  6.2.1 The special purpose DFG placement algorithm .................. 30
  6.2.2 Sugiyama placement algorithm .................................. 30
  6.2.3 Extending DFGREAD with the Sugiyama placement ............. 31
6.3 The interface between DFGREAD and ESCAPE ..................... 34
6.4 Translation of a dfggraph structure into keyword commands .......... 35

7 The tokenflow simulator ................................................ 36
7.1 Principle of tokenflow simulation .................................. 37
7.2 Tokenflow simulation algorithm .................................... 38

8 The simulation network data structure ................................ 40
8.1 Simulation network data structure .................................. 40
8.2 Token storage ......................................................... 40
In the Design Automation Section of the Department of Electrical Engineering at the Eindhoven University of Technology a program, ESCAPE has been developed for interactive entry and simulation of digital schematics. Two views are provided: a symbol view to design the graphical layout of modules and a network view to compose systems of symbols. These symbols either represent primitive modules or composite modules. Each primitive module has a behaviour description specified in a Lisp-like high level description language. A built-in event-driven simulator is used to validate the overall behaviour of the system. The results of the simulation are presented using animation techniques (see also [Fle93]).

A 'Data Flow Graph' (DFG) system representation has been developed to describe the behaviour of digital hardware. Its purpose is to serve as an intermediate format for circuits in architectural synthesis. Algorithms, described in hardware behaviour description languages such as VHDL, ELLA or HardwareC, are used to generate DFGs. Other steps in the silicon compilation process use DFGs as an intermediate system representation format. For each of these high level description languages a separate simulator would be necessary for the behavioural verification of the circuit. An alternative could be the usage of a DFG simulator for this verification. In order to develop such a simulator in ESCAPE, ESCAPE has to be extended with a graph view which makes it possible to handle all kinds of graph models.

A DFG simulator will also help creating a better understanding of this new specification level. It furthermore could assist in discussions on the actual DFG semantics.

This report describes the development of a DFG simulator within the context of ESCAPE.
1.1 Data flow graph simulation

In the DFG model operations are presented as nodes. Directed edges are used for the transport of data between nodes. Values on edges are modeled as tokens. A node can be executed if a token is present on each input edge of the node. Because two or more nodes can be executed at the same time, the execution order of the DFG's nodes, isn't deterministic. In this case a choice has to be made which node has to be executed first. So in order to simulate a DFG, some restrictions have to be made in the DFG model.

To simulate a DFG in ESCAPE, ESCAPE has to be extended with a graph view for displaying and editing of graphs. Next, the DFG simulator has to be implemented in ESCAPE. The simulator is based on the tokenflow principle and uses animation for the presentation of results. In order to place the token flow simulator in its context and to get more background material about simulation methods, a literature study has been performed (see [Fis73], [Jia90], [Hoo79], [Hoo82], [Hoo86], [Sme88] and [Shu78]).

ESCAPE is written in the C language and the User Interface has been built using X Windows¹ and the OSF/Motif toolkit².

1.2 Contents of the report

This report can roughly be split into two parts. The global contents of these two parts is listed below.

Part I Implementation of ESCAPE's graph view.

- Description of the DFG format.
- Development of a so called graphtype definition language (GDEF). With this format the different kind of nodes and edges can be described. E.g. the port specification and graphical description are part of a node type specification.
- Extending the graphical data structure of ESCAPE for graphs.
- Extending the network data structure of ESCAPE for graphs.
- The reading and interpretation of the textformat of DFGs and other graphtypes.
- Interactive editing of a graph.

Part II Development of the tokenflow simulator.

¹X Window System is a trademark of MIT
²OSF/Motif is a trademark of Open Software Foundation, Inc.
Introduction

- The principle of tokenflow simulation.
- Extending the network data structure for simulation purposes.
- The visualization of activity during simulation.
- The manipulation of tokens.
- Implementation of the behaviour of nodes.
- DFG and CTG (controlled DFG) simulation.
- Simulation of subgraphs.
- Mixed-level mode simulation of graphs with ESCAPE's discrete event simulator.
- Rewriting symbolic token expressions.
Chapter 2

Data flow graphs

This chapter describes the data flow graph (DFG) representation used for the synthesis and verification of integrated circuits, which can be derived from a behavioural level description.

2.1 Position of the DFG format

The development of the DFG standard started at the Eindhoven University of Technology back in 1986. After a few years, in 1990, the format and its semantics were adopted by the European ASCIS\(^1\) project, situated at seven universities and research institutes throughout Europe.

The DFG format is intended as intermediate format between user oriented interfaces (languages, schematics) and synthesis and verification tools. It also serves as an inter­changable format between different machines or sites.

The intended usage of the DFG format is illustrated in figure 2.1.

Due to the lisp-like structure of the DFG textformat, individual tools and sites can add information to it, without disturbing other tools who don't know about these additions. The format might be used throughout the synthesis process, by repeatedly annotating the graph with results of individual tools. This leads to a schema as in figure 2.2. For a complete specification of the semantics and textual format of the DFG standard I refer to [Eij91].

---

\(^1\)ESPRIT Basic Research Action 3281
2.2 DFG semantics

Earlier has been mentioned that a DFG consists of nodes and directed edges. These edges can contain tokens for storage of values. Execution of a node is done by taking tokens from incoming edges and after execution, putting the resulting tokens on the correct outgoing edges. Execution can only take place when tokens are present on the proper incoming edges. The execution order of the nodes in the graph is constrained by the partial ordering of the nodes as defined by directed edges.

It is required that only one incoming edge is connected to an input port. Output ports can have several outgoing edges connected to it. If execution of a node results in a token on an output port, this token is copied onto every outgoing edge connected to this port.

In the next sections a global overview of the DFG node types will be presented. A complete specification of the various node and edge types can be found in appendix A.1.

2.2.1 Operation nodes

Operations can be arithmetic, like *, -, +, ++ or boolean like <, = or more complex functions. The DFG format also provides for nesting of graphs in the same way as procedures in normal programming languages. The instantiation of an other graph is performed by using the name of the node type, which refers to the name of the corresponding subgraph. The port names used in the instantiation, correspond with the node names of the input and output nodes in the subgraph. An operation node always waits for execution until an input token is available on each input port.
2.2.2 Input and Output nodes

Each graph requires at least one node of type input, and can have one or more nodes of type output. Input nodes have no input ports and one output port. Output nodes have no output ports and one input port. If the graph would be instantiated elsewhere as an operation, the names of these input and output nodes define the port names of the operation.

2.2.3 Constant node

Nodes of type constant are nodes which can generate a constant token at their single output port. To indicate when such a token is to be produced, these nodes have one input port.
2.2.4 Branch and Merge nodes

The branch and merge nodes are necessary for building algorithmic constructions like if...then...else, or case...of.

A branch node has two input ports: a 'control' and a 'data' port. It can have two or more output ports with names '0', '1', and up. If a branch node is executed, the token on the 'data' port is transferred to the output port with the name indicated by the token taken from the control port.

Merge nodes are dual to branch nodes. They have two or more input ports with names '0', '1', and up, one 'control' input port and one output port named 'out'. If a merge node is executed, a token is passed from the input port with the name indicated by the token taken from the control port, to the 'out' port. The execution rule for a merge is slightly different from all other node types. A merge node can already execute when a token is present on the control port and the selected input port. Thus execution of a node does not always require the presence of a token on all inputs!

2.2.5 Entry and Exit nodes

Exit and entry nodes are functionally identical to the branch and merge nodes respectively. However these nodes are used to build constructions, which could originate from while...do or for...do statements.

To allow a proper execution of these loops a special initialization procedure is required. Before the execution of a graph can start, all input nodes obtain one token and at the same time all entry nodes must obtain a token at their control port, selecting the input port for external data to enter the loop. If the graph is repeatedly executed for different sets of input tokens, the loop constructs must not be reinitialized for each input set: such tokens are automatically left after each loop termination.

2.2.6 Get and Put nodes

Get and put nodes are defined to provide a mechanism for communication protocols with the outside world and can appear anywhere in a graph.

Get and put nodes, which use one physical port are linked with sequence edges to define the order in which read and write operation should appear on the port. Sequence edges are functionally identical to data edges except that they are used to define a sequence
order of nodes. For link constructions the input port 'chain-in' and output port 'chain-out' are used. For data transfer a get node has a 'data' output port, and the put node a 'data' input port.

2.2.7 Array operations

For operations on arrays three node types are available:

The array node. If this node is executed, the array is being initialized. It has one input port 'chain-in' and one output port 'chain-out' for constructing a link with other array operators.

For initialization the DFG textformat provides an 'array-dim' statement, which defines the dimension of the array and its size in each dimension. The array can be initialized with constant values with the 'const-value' statement.

The retrieve node is used to read a data value from an array. For constructing a link it has one 'chain-in' input port and one 'chain-out' output port. Furthermore it has one 'data' output port and one or more input ports with names '0', '1' and up, for indicating which index is used.

The update node is used to write a data value in an array. The update node has the same ports as a retrieve node except that the 'data' port is of type input.

2.2.8 Other operations

The noop, nothing and construct nodes have one input port 'in' and one output port 'out'. After execution the token on the input port is passed to the output port.

The value and delay nodes are treated the same way. Except that the input port is called 'N-0' and the output port is called 'N-1'.

2.3 DFG examples

In order to illustrate DFGs three examples are presented in this section. The first example in figure 2.3 depicts an if-statement implemented in HardwareC and its corresponding DFG, which can be generated from the HardwareC description. The second example in figure 2.4 is an implementation of the the function $a^b$. The last example is an
implementation of the \textit{faculty} function and can be found in figure 2.5. In the examples can be seen how the various algorithmic structures are translated into DFG structures.

```
process _if(a, b, o)
  in boolean a, b;
  out boolean o;
{
  if (a < b)
    o = a - b;
  else
    o = a + b;
}
```

Figure 2.3: If statement in HardwareC and its corresponding DFG
process _loop(a, b, o)
  in boolean a, b;
  out boolean o;
  { 
    o = 1;
    while (b>0) {
      o = o * a;
      b = b - 1;
    }
  }

Figure 2.4: The function $a^b$ implementation in HardwareC and its corresponding DFG

process _fac(i, o)
  in boolean i;
  out boolean o;
  { 
    o = l;
    while (i > 0) {
      o = o * i;
      i = i - 1;
    }
  }

Figure 2.5: Faculty implementation in HardwareC and its corresponding DFG
Chapter 3

Extending ESCAPE for DFG simulation

For implementing a DFG simulator in ESCAPE, a new view has to be added to ESCAPE which is called the graph view. In order to implement this view the following has to be done. In order to define the graphical representation of graph types, a graphical definition language (GDEF) has to be designed for the specification of the graphical representation of node and edge types of a specific graph type. Also a parser has to be implemented for reading the textual format of a DFG and the interactive editing of graphs has to be implemented. The different methods of capturing a DFG are illustrated in figure 3.1. The development of this graph view is described in the first part of this report.

First, the graphtype definition language (GDEF) is described. Second, the graphical data structure of ESCAPE is explained and extended for displaying graphs. Besides the graphical data structure also a network data structure has to be built, which is used to store the connectivity of nodes and edges. Last, the interface between the DFG textformat and ESCAPE is described (the DFGREAD utility, to read and to write DFG and other graph types).

The second part of this report is about the tokenflow simulator. First, the principle of tokenflow simulation is described. Next, the extending of the graph network data structure for simulation purposes is described. Also is explained how tokens can be manipulated (placed/removed) by the user or by the execution of nodes.

The simulator can operate in two different modes:

**DFG mode** DFG stand-alone simulation.

**CTG mode** The simulation of a DFG graph is controlled by a CTG graph.
Furthermore, the DFG simulator can be used in the mixed-level mode simulation of ESCAPE. The ESCAPE simulator is a discrete event simulator. So a DFG with or without a CTG can be used as a module in an ESCAPE schematic. Each time the module has to be evaluated the tokenflow simulator is called and the graph is executed. After execution the resulting data values become valid on the output ports of the module and the discrete event simulator can continue again.
If we want to visualize graphs, the graphical representation of graphs has to be defined in order to display them. Graphs are constructed with all kinds of different node and edge types. The graphical representation of a graph is built using the graphical representation of these different types of nodes and edges. The specification of the graphical representation of the node and edge types of a specific graph type is done by using a special purpose language called *graphtype definition language* (GDEF). This chapter describes the development of this GDEF language.

To access the graph definitions, ESCAPE is extended with an internal graph database which contains the graphical description of nodes and edges of the specified graph types. This database is constructed by processing GDEF files (e.g. 'dfg.gdef', which defines the node and edge type for the DFG).

Besides the graphical forms of nodes and edges, a GDEF file also contains information necessary for construction and simulation of graphs.

### 4.1 Requirements

Before we start designing the GDEF language, we have to perform an analysis of the language requirements.

The main requirements are the following:

- It has to contain all the necessary information for displaying a graph
- It has to contain other graph type specific information (e.g. for graph simulation)
- Easily extendable
Graphical simulation of Data Flow Graphs

- Easy to parse
- It has to be readable for humans
- If possible, it has to fit in the ESCAPE language

For the graphical definition of graphs the following items have to be specified.

Graph

- Name of the graph type
- The node types
- The edge types
- A number of settings:
  - Are ports used?
  - Is the graph directed?
  - The grid placing

Node

- Name of the node type
- Bounding object of the node (e.g. a circle)
- Fill color of the bounding object
- Name, type and position information of ports
- Other symbols such as e.g. text
- For interactive entry: the menu in which the node has to be placed
- For simulation of its behaviour: the specification of the name of the node execution function

Edge

- Name of the edge type
- The line style used (solid, dotted or dashed)
- The edge color
4.2 Development

For the development of the language I've used the tools LEX and YACC. LEX is used for the generation of a lexical scanner and YACC converts a context-free grammar into a parser, which uses the tokens generated by the lexical scanner. While parsing the recognized tokens, the GDEF database is constructed by the parser.

As a basis for the GDEF language I've chosen a lisp-like language of type LL-1. A simple recursive descent parser can parse these lisp-like languages.

Advantages of this language type are the following:

- No set of reserved words is required.
- The language is easy extendable.
- It is easy to parse.
- The ESCAPE language is also a lisp-like language. So it should not be a problem to include the GDEF format in the ESCAPE format.

The specification of the GDEF format can be found in appendix B.

4.3 Some examples

In this section some examples are presented to illustrate how the GDEF language is used. Figures 4.1, 4.2 and 4.3 present the examples of the GDEF descriptions of the input, add and branch node respectively.

The bounding object of a node can either be a circle or a polygon. The 'fill-color' statement indicates the fill color of the bounding object. The specification in which menu the node has to be placed is specified by the 'menu' statement (this is used for interactive entry of graphs). A port can be of type 'input', 'output' or 'inout' (input and output). If the port position is specified 'at-boundary' it means that outgoing or incoming edges point to the so called hotspot (position x=0, y=0) of the node. The edges begin or end respectively on the boundary object. It is also possible to specify a fixed port position relative to the hotspot. Additional symbols can be placed in the symbols list. The information specified by the 'function' statement is used for simulation purposes and specifies the name of the node execution function.
As can be seen in the example of the branch node type (figure 4.3), the branch node has one output port with a property variable. This means that the branch node can have several output ports depending on how this node is used in a graph.
Graphtype Definition Language

Figure 4.3: Branch node GDEF description and its graphical equivalent
In this chapter the graphical data structure of ESCAPE is explained. More specific, how graphical objects like lines, arrows or circles are stored and manipulated.

The next step towards the implementation of the graph view, is to extend this set of graphical objects with a \textit{node} and an \textit{edge} type and to define the functions to operate upon these new graphical objects for moving, deletion, drawing and so on.

Also has to be taken in account that all kinds of different node and edge types exist. Each type has its own graphical representation and other type specific parameters.

\section{The graphical structure of ESCAPE}

In ESCAPE a design is graphically represented by a object oriented data structure. A design is represented by a set of graphical objects. Each graphical object, no matter what type, has a \textit{GObject} data structure. This \textit{GObject} structure contains a pointer to a type specific data structure, which have names like \textit{GOBJLine}, \textit{GOBJCircle}, and so on. In these structures specific information is stored for drawing the graphical object (e.g. the ending points of a line or the circle radius).

Each different kind (class) of graphical object has its own set of functions for drawing, moving, and so on. Therefore the \textit{GObject} structure also contains a pointer to a structure called \textit{GObjectClass}. In this structure pointers to functions with names like \textit{DrawLine} or \textit{MoveCircle} are stored. The structure \textit{GObjectClass} also contains other class specific information (e.g. the name of the class). An advantage of using a class structure is that new types of graphical objects can easily be added. Because each different graphical object is represented by the same data structure (\textit{GObject}), it is possible to execute object’s type dependent functions using the same access function.

All these \textit{GObjects}, which all together graphically represent a design, are stored in a
region query data structure, which is implemented as a linked list of GObjects. Other data structure implementations are also possible. So, if a design has to be drawn, each GObject in this list has to be drawn. This implies that the draw function of each GObject has to be executed.

In figure 5.1 the relationship between the graphical data structures is illustrated.

![Graphical representation](image)

**Figure 5.1: The core graphical data structure of ESCAPE**

The core data structures to store graphics in ESCAPE (GObject, GObjectClass and as an example GOBJLine) are the following:

**GObject**

```c
typedef struct _GObjectRec {
```
Graphical simulation of Data Flow Graphs

typedef struct _GObjectClassRec {
    char *class_name;
    unsigned int obj_size;
    GObjectClass superclass;
    Bool inited;
    void (*class_init)();
    GObject (*create_func)();
    GObject (*copy_func)();
    void (*orientate_func)();
    void (*translate_func)();
    void (*multiply_func)();
    void (*delete_func)();
    void (*edt_name_func)();
    void (*size_func)();
    Bool (*locate_func)();
    void (*draw_func)();
    void (*plot_func)();
    void (*ileaf_func)();
    void (*select_func)();
    void (*save_func)();
    void (*new_save_func)();
    void (*load_func)();
    void (*rub_handlers)();
    void (*geo_func)();
    void (*undo_func)();
    GC gc;
    char *free_list;
    char *extension;
} GObjectClassRec;

typedef struct _GOBJLine {
    int x, y;
    int x_end, y_end;
    GC gc;
    int depth;
} GOBJLine;

Because of its class structure it is quite easy to add new graphical objects to this data structure.
5.2 Extending the graphical structure of ESCAPE for graphs

For the implementation of the graph view it has to be possible to display nodes and edges. This is made possible by extending the graphical data structure of ESCAPE with two new GObject types: GOBJNode and GOBJEdge.

The GOBJNode data structure has to contain the following items:

- Position information, the x and y coordinates of the hotspot (center) position of the node. The hotspot position is also used to connect nodes with edges.
- The graphical representations of the bounding object, additional symbols and optional ports. These representations are constructed by linked lists of GObjects (like lines or text).
- A pointer to the node definition in the GDEF database. This pointer can be used to access other node type information.
- A pointer to the network data structure of the node. This network data structure defines the interconnection between nodes, edges and ports.

The corresponding data structure:

```c
typedef struct _GOBJNode {
    int x;
    int y;
    struct _GDEF_NODE *def; /* only, if non-default node */
    GObject bound_obj;
    GObject symbols;
    GObject pins; /* only, if pins are visualized on screen */
    void *nref;
    int depth;
} GOBJNode;
```

The GOBJEdge data structure has to contain the following items:

- The position of the hotspot of the source and destination nodes. This makes it possible to place the edge before the source and destination nodes are placed.
- The graphical representation of the edge. This representation is stored as a linked list of GObjects.
- A pointer to the edge definition in the GDEF database.
- A pointer to the network data structure of the edge.
Graphical simulation of Data Flow Graphs

The corresponding datastructure:

```c
typedef struct _GOBJEdge {
    int x1; /* origin - node position */
    int y1;
    int x2; /* destination - node position */
    int y2;
    struct _GDEF_EDGE *def; /* only, if non-default edge */
    GObject symbols; /* symbols to draw the edge (if not default) */
    GObject bezier;
    void *nref;
    int depth;
} GOBJEdge;
```

Also functions have to be implemented to operate upon these new GObjects. The following functions are implemented for the node GObject:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitClass</td>
<td>Initialization of the class</td>
</tr>
<tr>
<td>CreateNode</td>
<td>Create a new node</td>
</tr>
<tr>
<td>CopyNode</td>
<td>Copy a node</td>
</tr>
<tr>
<td>TranslateNode</td>
<td>Translate a node</td>
</tr>
<tr>
<td>DeleteNode</td>
<td>Delete a node</td>
</tr>
<tr>
<td>LocateNode</td>
<td>Locate a node</td>
</tr>
<tr>
<td>DrawNode</td>
<td>Draw a node</td>
</tr>
<tr>
<td>PlotNode</td>
<td>Make a postscript plot of a node</td>
</tr>
<tr>
<td>IleafNode</td>
<td>Make an ileaf plot of a node</td>
</tr>
<tr>
<td>NodeHandlers</td>
<td>Add node handlers for editing</td>
</tr>
<tr>
<td>UndoNode</td>
<td>Undo the last operation done on a node</td>
</tr>
</tbody>
</table>

For the edge the same set of functions has been implemented.

As an example I will explain how to draw a GObject. This is done by using the function `DrawGObject`. This function will call the drawing function of which a pointer to it can be found in the GObjectClass record. In this way a GObject type specific drawing function will be executed.

For drawing a complete design one has to call the DrawGObject function for each GObject in the design.

5.3 The network representation of a graph

Besides in a graphical data structure, a graph is also represented in a network data structure in which the interconnection between nodes, edges and ports is stored. Each design is represented by a DESIGN structure which contains a linked list of graphs in
the design. All designs in ESCAPE are stored in a linked list also and can be accessed by a pointer called 'designList'. In figure 5.2 the graph network data structure is presented.

Each graph is represented by a GRAPH structure. Nodes and edges are stored in the structures GRAPHNODE and GRAPHEDGE respectively. All nodes and edges in a graph are stored in linked lists.

Interconnections between nodes and edges are made using ports (GRAPHPIN structure). All ports of a node are stored in a linked list. Each port can have several edges attached to it. These edges are stored by using a linked list of a structure called LINK. Each LINK structure contains a pointer to an edge.

It is also possible to find the origin and destination node of an edge. The GRAPHEDGE structure contains two pointers to GRAPHPIN structures. Each GRAPHPIN also has a pointer to its node. In this way it is possible to walk through the graph in all directions. It is also possible to store graphs which don’t use pins. In this case the two pointers (source and destination) in a GRAPHEDGE structure point directly to a node.
The graphical and network data structures of a graph are cross referenced. Each node and edge in the network representation contains a pointer to its corresponding GObject. As mentioned earlier the GOBJNode and GOBJEdge data structures contain each a pointer to respectively a GRAPHNODE and GRAPHEDGE.

An advantage of keeping graphical and network data separate, is that different network data structures can be used, while the graphical data structure still remains the same. A disadvantage could be the overhead in pointers.
Chapter 6

Reading the graph textformat

For reading graphs in ESCAPE, a program called DFGREAD has been developed. It can parse the lisp-like language of the DFG textformat. First this format was intended to be used for describing data flow graphs (DFG). Later, the DFG textformat was extended for describing control graphs (CTG), network graphs (NWG) and their inter-relations as well. This new extended format is now called the ASD textformat.

After parsing the textformat, each graph is being placed by a placement algorithm. This placement algorithm derives a placement of the nodes in the x-y plane for a given graph structure. Default a special purpose placement algorithm is used which is especially designed to place DFGs. This placement takes in account the special structure of DFGs. Also the Sugiyama graph placement algorithm has been implemented (see [Sug81]). This placement can be used for a larger class of directed graphs.

For clarification, the processing of graphs by DFGREAD is illustrated by figure 6.1.

DFGREAD is an external program. So the next step will be the transmission of the placed graph information to ESCAPE. Communication between DFGREAD and ESCAPE is established by using a so called inter-tool protocol, which is provided by ESCAPE. This protocol uses keyword commands for information transfer between ESCAPE and other tools. The main advantage of using external programs via a communication protocol, is that ESCAPE's functionality will be increased without increasing the overall size and complexity of ESCAPE.

6.1 Parsing ASD files

A parser for the DFG textformat had already been developed. The parser with its data structures forms the basis of the DFGREAD tool and has to be extended for parsing CTG, NWG and the inter-relation between graphs.
First the construction of the parser will be explained. For parsing new graph types, the parser will be extended with a number of new production rules.

### 6.1.1 Construction of the parser

The DFG parser is built in a very inventive way by using the macro-expansion feature of the C language.

To give an impression how this is done, the C macros for building the parser are listed below.

```c
/********** set of defines, for building a compact looking parser **********/
#define Newtoken ( (token==LEX_EATEN&& (token=dfg_lex())), \ 
 token!=LEX_EOF&& token!=LEX_CLOSE)
#define Rule static int
#define Is () { int result;
#define Form(keydef,flags) ;if (token != keydef) result 
} char *flag=flags; int thisform=token; \ 
 { result=1; token=LEX_EATEN; { 
#define InOrder ;)if (result 
#define OptInOrder );if (result && Newtoken) {result= 
#define NoOrder ;)while (result && Newtoken) {result = 
#define Code ;if (result) 
#define EndForm );result=check_listend( thisform, result, flag); \ 
 } return( result);)
#define FormO(keydef) ;if (token != keydef) result = 0; else
```
Reading the graph textformat

{ int thisform = token; 
  result = 1; token = LEX_EATEN; 
  #define EndFormO 
} 
result = check_listend( thisform, result, "" ) 
  ) return( result ); 
/* Usage of flags: bit 0: (value 1) argument has appeared in input */ 
/* bit 1: (value 2) argument should appear at most once */ 
/* bit 2: (value 4) argument should appear at least once */ 
/* These bits 'or' in the flag to show different situations */ 
#define Flag(arg,i) ( (flag[i] & 3) != 3 && (arg) ? (flag[i] = 1) : 0 )

These macros can be used as building blocks for specification of production rules in C. An example of this kind of construction is given below.

Rule F_graph Is Form! K_graph, '422*)
InOrder is_identifier( )
  Code { graph = CALLOC_STRUCT( DFG_GRAPH) ; 
    graph->name = new_string( dfg_lex_word); 
    graph->view = view; 
    hash_add( graph_hash, graph->name, graph); 
    node_hash = graph->node_hash = hash_create( "node hash", 307, 0); 
    edge_hash = graph->edge_hash = hash_create( "edge hash", 307, 0); 
    writtenp = &(graph->written); 
    } 
  NoOrder F_edge() || 
  Flag( F_node(), 0 ) || 
  Flag( F_status(), 1 ) || 
  Flag( F_bbox(), 2 ) 
EndForm

In the example you can see that during parsing the textfile, the graph is being stored in a data structure.

The syntax of this rule is the following:

<graph> ::= "(" "graph" <identifier> (<node>)+ (<edge>)+ [<status>] [<bbox>] ")".

Within the graph context a node description should appear one or more times. Edge descriptions can appear zero or more times. The description of the bbox (boundingbox) and the status are optional and should appear at most once.

Note that the syntax notation given above isn't correct because it specifies that all node descriptions should come first before e.g. the status description. In the corresponding C macro structure it is also possible that nodes, edges, status and the bounding box can appear in any order!
6.1.2 Extending the DFG parser for ASD files

The DFG parser now has to be extended in such a way that it can also process control (CTG), network (NWG) and the inter-relations between graphs. The first two graph types look quite the same as DFGs. So only a few extra production rules have to be added and the rule for parsing a node description has to be extended.

Parsing control and network graphs

The new node rule is the following:

```plaintext
/** extended node rule of the 'old' DFG parser **/
Rule F_node Is Form( K_node, "$122222222222"
    InOrder is_identifier()
    Code { if (!hash_get( node_hash, dfg_lex_word, &node))
        node = new_node( node_hash, dfg_lex_word);
        else if (!node->type)
            parse_err( "Node name '%s' not unique!
            node->name);
        varname = &node->varname;
        delay = &node->delay;
    )
    NoOrder Flag( F_type_node(), 0 ) ||
    Flag( F_const_value(), 1 ) ||
    Flag( F_in_edges(), 2 ) ||
    Flag( F_out_edges(), 3 ) ||
    Flag( F_selection_list(), 4 ) ||
    Flag( F_varname(), 5 ) ||
    Flag( F_src_line(), 6 ) ||
    Flag( F_schedule_time(), 7 ) ||
    Flag( F_network_width(), 8 ) ||
    Flag( F_network_parameter(), 9 ) ||
    Flag( F_array_dim(), 10 ) ||
    F_position()
}
EndForm

The additional rules which make it possible to parse control and network graphs are:

```plaintext
/**** control-view rule(s) ****/
Rule F_control_view Is Form( K_control_view, '4')
    Code  view = CONTROL;
    NoOrder Flag( F_graph(), 0)
EndForm

/**** network-view rule(s) ****/
Rule F_network_view Is Form( K_network_view, '4')
    Code  view = NETWORK;
    NoOrder Flag( F_graph(), 0)
EndForm

Rule F_network_width Is Form( K_width)
    InOrder is_integer( &node->width))
EndForm
```
Also the starting rule of the parser has been extended so the control_view and network_view rules are evaluated.

Parsing the inter-relation specification between graphs

The inter-relation specification between graphs describe the links between nodes of different graphs.

First of all it describes which graphs are associated. Second, it describes which nodes of the DFG, CTG and NWG are linked to each other.

These link between nodes of different graphs are necessary for simulation of CTGs. If a CTG nodes is executed, the corresponding DFG nodes are executed.

For parsing these inter-relation specifications, a whole new set of production rules has to be constructed.

The syntax of the link view is the following.

```plaintext
<link_view> ::= "'link-view' ("graph_link")+ '").
<graph_link> ::= "'graph-link' <identifier> <data_flow_name> <control_name> <network_name>
<node_link>++
<data_flow_name> ::= "'data-flow' <identifier> '").
<control_name> ::= "'control-flow' <identifier> '").
<network_name> ::= "'network-flow' <identifier> '").
<type_node> ::= "'node-link' [<type_node>] [<data_flow>] [<control>] [<network>] '").
<data_flow> ::= "'data-flow' <identifier> '").
<control> ::= "'control' <identifier> '").
<network> ::= "'network' <identifier> '").
```

These rules can also be written with the use of the C macros. The 'type_node' rule is used from the 'old' DFG parser. The C macro versions of these rules can be found in appendix C.
6.2 Graph placement

After parsing the graphs and building the corresponding graph data structure, each graph is being placed in the x-y plane by a graph placement algorithm. First, two placement algorithms provided are explained. Next, the improvement of the Sugiyama placement algorithm for placement of loop contructions is treated. Also is explained how DFGREAD is extended with the this placement algorithm.

6.2.1 The special purpose DFG placement algorithm

The special purpose DFG placement algorithm consists of two phases:

Phase 1 - Vertical placement of nodes A depth-first search algorithm is used to assign nodes to a level (vertical position). If an edge crosses one or more levels, this edge will be broken into segments, and dummy nodes are introduced. These dummy nodes are treated the same as the other nodes. The usage of dummy nodes will prevent edges to intersect with nodes. In the depth-first search algorithm, all branch nodes attached to the same control node will be treated as one node. The result of this will be that these branch nodes will be placed on the same level. The same procedure is also used on merge, entry and exit nodes.

Phase 2 - Horizontal placement of nodes The input nodes are placed in random order. For each node a so called gravity value (a floating point value) will be calculated. The gravity value of a node is equal to the average gravity value of its predecessor nodes. In this way each node will be placed at the average horizontal position of its predecessor nodes. If a node has no predecessors (input nodes), the node will get a gravity value equal to zero. By sorting the nodes on each level by gravity value, a horizontal order of nodes on a level is derived. All nodes have a minimum distance to each other. If this distance is too small, the spacing between the two nodes will be adjusted to this minimum distance. This horizontal placement starts at the top level and moves down. The second pass starts at the bottom and moves up to the middle of the graph. The last pass will place all nodes in upper half of the graph by starting at the middle and move up. The nodes in the lower half will be placed by also starting at the middle and move down.

6.2.2 Sugiyama placement algorithm

The Sugiyama layout algorithm has three phases. (See also [Sug81] and [Row87]).
This first phase assigns nodes to levels. The second phase sorts the nodes on each level to minimize the number of edge crossings. The third and final phase fine tunes the positioning of nodes and routing of edges to make the layout easier to understand.

**Phase 1** Nodes are assigned to levels in the first phase so that each node is on a level below its predecessors. The first step is to determine which nodes have no predecessors. These nodes are placed at the top of the graph by assigning them to level 0. Each other node gets the maximum value of the level numbers of its predecessor nodes plus one.

Next, long edges that connect nodes that are more than one level apart, are broken. These long edges are broken into segments, each of which goes between adjacent levels, and dummy nodes are inserted. Dummy nodes and the segments of long edges are treated by the rest of the algorithm the same way as normal nodes and edges. The reason for breaking up these long edges is to allow edges to bend at each level, thereby providing flexibility in positioning of the two nodes connected to the long edges and permitting a more compact graph layout.

**Phase 2** The second phase of the algorithm makes multiple passes over the graph, changing the order of nodes on each level to minimize the number of edge crossings. The first pass begins at the top level of the graph and moves down. The second pass begins at the bottom level and moves up. Subsequent passes alternate between top-down and bottom-up passes until the algorithm terminates. The algorithm can terminate when all edge crossings have been eliminated or after a fixed number of passes.

**Phase 3** The third phase of the algorithm fine-tunes the layout. The first two phases determine the level assignment and the order of nodes on a level. The fine-tuning phase determines the actual position of the nodes on the x-y plane. First the nodes are evenly distributed on a level and levels are separated by a uniform distance. The second step makes several passes over the graph to straighten long edges.

### 6.2.3 Extending DFGREAD with the Sugiyama placement

I have taken an implementation of the Sugiyama algorithm from an other application called 'GraphEd' (See [New88]), which is an interactive editor for graphs and graph grammars.

The Sugiyama algorithm works upon its own internal graph data structure (sgraph). So the easiest solution will be to copy the graph structure of DFGREAD (dfggraph) to the
Sugiyama sgraph representation. The next step will be to perform the Sugiyama placement and finally to copy the newly added information of the sgraph into the dfggraph.

The Sugiyama algorithm will add coordinate information of nodes in the sgraph. Also new nodes called 'dummy' nodes are added. The reason why these dummy nodes are introduced will be explained later.

To make it easier to copy the new information back from the sgraph to the dfggraph structure, the nodes in both structures have cross references to each other.

Sugiyama placement is done by the following steps:

1. Create a sgraph from a dfggraph
2. Execute the Sugiyama placement algorithm
3. Copy back the coordinate information of each node in the sgraph and create also the introduced dummy nodes and dummy edges in the dfggraph

Extending the Sugiyama algorithm for loop placement

The Sugiyama placement is not intended to be used for graphs with loop structures. In DFGs however, loop structures are a common phenomena and therefore the Sugiyama placement should be improved to be able to place graph with loop structures.

Making this improvement implies tackling two problems:

1. Make the vertical placement (level assignment) of nodes deal with loop structures.
2. Improve the horizontal placement for placement of loop structures.

Ad.1 A new level assignment algorithm

The first step in graph placement is the construction of a vertical order in the graph. This is done by the assignment of each node to a level.

The Sugiyama algorithm first detects the existence of loop structures in a graph. If such a structure is recognized the graph isn’t placed and an error is generated.

Level placement of non-loop graphs is simply done with the following recursion.
set_level(Snode n, int level)
{
    if (level(n)<=i) {
        level(n) = level;
        for_all_outgoing_edges (n,e)
            set_level(e->destination-node, level+1);
    }
}

Initially, all level values of nodes will be set to zero. Level placement is done top-down
and starts by calling the set_level function for all nodes with no incoming edges with an
argument a level equal to zero.

If a graph with a loop is placed by this algorithm, this recursion would result in an
endless loop. So another algorithm for level placement should be invented.

A characteristic of a vertical placement is that the level of a node is defined as the
maximum value of the levels of its predecessor nodes plus one.

With this knowledge we can rewrite the set_level recursion:

set_level(Snode n, int level)
{
    int max = -1;
    Sedge e;

    mark (n);
    for_all_incoming_edges(n,e) {
        if (not marked (e->source_node)) set_level(e->source_node);
        if (level (n) > max) max = level(n);
    }

    level(n) = max + 1;
}

Starting the level placement is done bottom-up by calling the set_level function for
all nodes with no outgoing edges. This is an implementation of a depth first search
algorithm.
Stand alone nodes (nodes with no edges attached to it) are automatically placed in the same level.

**Ad.2 Horizontal placement of loop structures**

Originally the placement algorithm was intended to be used for non-loop graphs. So we have to fool the algorithm by using a trick. Loop structures can be recognized by a backward 'jump' of an edge. So if an edge goes up, which means the level of the source node is higher than the level of the destination node, the edge has to be reversed. In this way the graph is made loop-free.

After placement these reversed edges have to be reversed again to get a proper graph again.

### 6.3 The interface between DFGREAD and ESCAPE

After a graph is placed, the graph information has to be sent to ESCAPE. This is done by using the *inter-tool protocol* of ESCAPE. Through this protocol, instructions for ESCAPE can be sent using ascii keyword commands.

The following keyword commands are used:

<table>
<thead>
<tr>
<th>Keyword command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph &lt;name&gt; &lt;type&gt;</td>
<td>add a new graph</td>
</tr>
<tr>
<td>end-graph</td>
<td>ends a graph description</td>
</tr>
<tr>
<td>add-node &lt;x&gt; &lt;y&gt; &lt;name&gt; &lt;type&gt; [more]</td>
<td>add a new node</td>
</tr>
<tr>
<td>property array-dim &lt;string&gt;</td>
<td>sets the array dimension of 'array' nodes</td>
</tr>
<tr>
<td>property const-value &lt;integer&gt;</td>
<td>sets the value(s) of 'const' and 'array' nodes</td>
</tr>
<tr>
<td>property name &lt;string&gt;</td>
<td>specifies the name of a subgraph</td>
</tr>
<tr>
<td>end-add-node</td>
<td>end node description</td>
</tr>
<tr>
<td>add-edge &lt;x1&gt; &lt;y1&gt; &lt;x2&gt; &lt;y2&gt; &lt;name&gt; &lt;type&gt; [more]</td>
<td>add an edge between nodes on the two positions</td>
</tr>
<tr>
<td>add-line &lt;x1&gt; &lt;y1&gt; &lt;x2&gt; &lt;y2&gt;</td>
<td>add a line</td>
</tr>
<tr>
<td>add-arrow &lt;x1&gt; &lt;y1&gt; &lt;x2&gt; &lt;y2&gt;</td>
<td>add an arrow</td>
</tr>
<tr>
<td>orig-port &lt;name&gt;</td>
<td>specifies origin port name of edge</td>
</tr>
<tr>
<td>dest-port &lt;name&gt;</td>
<td>specifies destination port name of edge</td>
</tr>
<tr>
<td>end-add-edge</td>
<td>ends an edge description</td>
</tr>
</tbody>
</table>

The option 'more' is used to indicate that more information has to be received. In this case end-add-edge/end-add-node command should be used to end the add-edge/add-node command.
6.4 Translation of a dfgggraph structure into keyword commands

The last step will be the information transmission of the graph that has been placed, using inter-tool protocol commands.

First, the graph name and type are being transmitted by using the 'graph' command.

Then all nodes in the graph are sent. This is done by using the 'add-node' command. There are nodes which have other information that also has to be sent. For example, an array node has a dimension specification and can have initial values. This kind of information is sent by using the 'property' command. A node description is completed by the command 'end-add-node'.

Next, all edges are sent. This is done by sending all incoming edges of each non-dummy node in the graph. An edge begins with an 'add-edge' keyword command. Next the origin and destination ports are transmitted. An edge can consist of more than one segment. In the graph structure such a construction of edge segments is stored by using so called dummy nodes. A dummy node is nothing more than an edge point. Thus sending an edge is done by traversing the list of edges in the graph data structure from a node until a non-dummy node has been reached. Each edge segment has to be sent either as an arrow or a line. This is done by the keyword commands: 'add-arrow' and 'add-line'.

At the begin and the end of an edge the intersection of the edge with the node bounding object is calculated.

A sample of output generated by DFGREAD is listed in figure 6.2.

```
... add-node 30000 30000 N-8 const
add-node 30000 90000 N-9 radd
add-edge 15000 0 0 30000 E-7 source more
orig-port out
dest-port N-0
add-arrow 13500 3000 2683 24633
end-add-edge
add-edge 50000 0 30000 60000 E-10 data more
orig-port out
dest-port N-1
add-arrow 50000 30000 33328 55007
add-line 50000 6000 50000 30000
end-add-edge
...
```

Figure 6.2: An Example of a sample of output generated by DFGREAD
The second part of this report will describe the implementation of the tokenflow simulator in ESCAPE.

The following subjects will be treated:

- The token flow simulation principle.
- Extending the network data structure for simulation purposes.
- A mechanism for token storage.
- Token manipulation:
  - Interactively by the user.
  - By functions used in node execution functions.
- How to visualize activity.
- DFG and CTG simulation modes.
- Simulation of subgraphs.
- Mixed-level simulation of the tokenflow simulator with ESCAPE's discrete event simulator.
- Interface with the REWRITE program, for rewriting symbolic token expressions.

The main goals of such a simulator will be the following:

- To create a better understanding of the DFG standard. The DFG simulator can be used for education.
The tokenflow simulator

- As a tool for analyzing DFGs. E.g. verifying whether or not a graph is safe. Safe graphs are graphs which have at most one token on each edge during execution.
- To simulate a whole range of HDLs by using the DFG standard as an intermediate format for simulation. This makes separate simulators for HDLs superfluous.
- To use a DFG for specification of the behaviour of modules. These modules can be used in ESCAPE simulations and are evaluated by using the DFG simulator.

7.1 Principle of tokenflow simulation

In order to simulate a DFG, the behaviour of the DFG must be simulated. This behaviour is defined by the behaviour of the nodes and the relationships between nodes determined by edges.

Information in a graph is modeled by tokens. A token is a basic structure in which a data value can be stored. Edges are used for the storage of tokens and operate as FIFO (First In First Out) queue. The definition of the behaviour of a node is defined in terms of token activity. Tokens can be absorbed, passed or generated by a node.

For example, a node for performing an addition is implemented as a node with two input ports and one output port. When simulating this node the tokens on the edges connected to the input ports are taken. A new token is generated with a value equal to the addition of the two input values. This new token is passed via the output port on the attached output edges. This example is illustrated in figure 7.1.

![Before execution](image1.png)

![After execution](image2.png)

Figure 7.1: Numeric execution of an add node

The above example describes the numeric execution of an add node. Other types of execution are also possible, e.g. symbolic. Symbolic node execution requires the input
tokens to be of type symbolic (strings). The resulting token will also be a symbolic one.

### 7.2 Tokenflow simulation algorithm

A DFG is executed in cycles. In the initial state at least one token must be present at all output edges of each input node. By the presence of tokens, some nodes are executable. The execution of these executable nodes is done by performing a so called cycle step. Simulation of a graph is done by repeated execution of cycle steps and is terminated if no nodes can be executed anymore.

For the simulation administration two event lists are used. An event is used to administrate that a node has to be executed.

**CEL** Current Event List. In this list nodes are places which are to be evaluated in one cycle step. This list can be considered a working list for the token flow simulator.

**FEL** Future Event List. In this list nodes are placed which are to be evaluated in the next cycle step.

In the following cases a node is checked whether or not it is ready for execution:

- An edge connected to an input port of the node has got a new token placed on it. This can be done interactively by the user or as a result of the execution of a node.
- The node is executed and may also be executable in the next step.

Normally a node can be executed when all incoming edges contain at least one token. For these nodes a so called empty value has been introduced. If no incoming edges contain tokens this value has its maximum value, which is equal to the number of ports. Each time a token is placed on an empty incoming edge of a node, this empty value is decreased by one. So a node is executable when this empty value has become zero.

Also nodes exist with different execution conditions. For example, the execution rule for a merge type node is slightly different from all other node types. This node can already execute when a token is present at the control port and the port, selected by the value of the token at the control port.

The graph simulation algorithm is the following:
The **tokenflow simulator**

**RESET** Reset the graph to initialize simulation
- Empty current event (CEL) and future event lists (FEL).
- Remove all tokens from the edges.
- Initialize all nodes for execution.
  - Initialize the 'empty' counters.
  - Place initial tokens on the input ports of some nodes if necessary.

**PLACE TOKENS** Tokens can be placed in the graph by the user or automatically.

**CYCLE STEP** Perform a cycle step.
- Transfer all events from the FEL to the CEL.
- Evaluate all events in the CEL.

**REPEAT?** repeatedly perform cycle steps until no events are left in the FEL.

**END** No nodes can be executed any more. The results can be read in the graph.

The steps in the simulation algorithm are depicted in figure 7.2.

![Figure 7.2: The graph simulation algorithm](image)

It is also possible to let the user control the 'repeat' step of the algorithm. The user can 'single step' the graph and it is possible to add and remove tokens between two cycle steps. It is also possible to execute nodes interactively, which allows the user to determine the execution order of the nodes.

Normally, DFGs are generated from a HDL like HardwareC. If this generation is done properly, the created graph will always execute properly (safeness), i.e. no tokens (excepts control tokens for loop constructions) will remain in the graph after execution. During the execution of a cycle step the execution order of the nodes will be of no effect on the final result.
Chapter 8

The simulation network data structure

The network data structure which is used to store graphs doesn't provide storage used for simulation of graphs. Therefore it has to be extended for simulation purposes. First, it is described how the data structures to store graphs, nodes and edges are extended. Second, a new data structure is introduced for the storage of tokens.

Also node types exist which require more items to be stored. Such nodes are e.g. const (constant value) and array nodes (array contents).

8.1 Simulation network data structure

In order to simulate a graph, a new network data structure has to be designed which is compatible with the existing one. Meaning that functions which operate on the existing network data structure can still be used on the new simulation network data structure. Figure 8.1 illustrates the new network data structure.

The compatibility requirement is met by using an overlay technique: the existing graph network structures is placed at the top of the new structures. For example, a variable of type TSIMNODE can also be used as if it were of type GRAPHNODE.

8.2 Token storage

In the graph model tokens can be stored on edges. This storage is done by putting the tokens, situated on a specific edge, in a linked list. Each time a token has to be taken from an edge, the first token in the list is removed and returned. New tokens placed on
The simulation network data structure

Figure 8.1: Extending the network data structure

the edge are put at the end of the token list. In this way a token list operates as a FIFO (First In First Out) queue.

Tokens can be seen as data packages, which can carry all kind of data. Data types which are provided are the numeric and symbolic type. A numeric data value is stored as an integer and a symbolic data value is stored as a string. Symbolic tokens are used to store names of variables. These symbolic tokens make it possible to simulate a graph in a symbolic way. Numeric simulation is done by using numeric tokens.

The basic structure for token storage is listed below. The type field indicates how the union value should be used. A metaphoric representation of the token types is illustrated in figure 8.2. Tokens are represented as carriages which can contain different kinds of loads.

Figure 8.2: Metaphoric representation of tokens

**TSIMNODE**

typedef struct _TSIMNODE {
    struct _GRAPHNODE network;
    int empty;
    void (*exec_node)();
} TSIMNODE, *TSIMNODE_PTR;

**TSIMTOKEN**

typedef struct _TSIMTOKEN {
    int type;
    union {
        int number;
        char *symbol;
    } value;
    struct _TSIMTOKEN *next;
} TSIMTOKEN, *TSIMTOKEN_PTR;
Chapter 9

Token manipulation

The key issue in token flow simulation is the manipulation of tokens. Three ways to manipulate tokens are provided by the token flow simulator:

- Interactive editing done by the user.
- Reading and writing a token distribution file.
- Node execution.

In this chapter these three types of token manipulation will be described.

9.1 Interactive token editing

At all times the user must be able to edit the tokens in a graph. This editing can be done interactive just by clicking on edges or nodes.

9.1.1 Placing of tokens

The user can place and remove tokens on edges in two ways. The first way is to select the 'place token' option in the menu and to click on an edge. This will popup a window for entering the token value. The value type is determined by the token mode which can be changed by selecting 'options...' in the menu.

In DFGs, it is common to attach several edges to one output port of a node. Placing a token on each of these edges is a very time consuming business. Therefore another token placement feature is added. All edges connected to a single port can be supplied with a token in one single action. The user has to select the 'Set port value' option in the
menu and can click on a node. A popup menu will appear in which the user can indicate on which port the token has to be placed. Next, the value can be entered. If a node has one port only, this port is automatically selected and the port selection will be skipped.

Presence of tokens is visualized by giving the edge a different color. Depending on the so called *tokenviewer mode*, it is also possible to display either the first token on the edge or the number of tokens on the edge. This is done by using a so called *tokenviewer*. This tokenviewer which will be drawn whenever an edge, with one or more tokens, is drawn.

### 9.1.2 Removing tokens

Removing tokens can be done by selecting the menu option 'Remove token'. The user now can select an edge by clicking on one. The first token will be removed from this edge.

### 9.2 Reading and writing a token distribution file

Instead of placing tokens in a graph for each simulation run again, an option has been provided for reading and writing tokens. The save format is very simple and is based on keyword commands. The BNF rules of the format are the following:

**Starting rule**

```
<distribution> ::= <edge> | <node> | <comment>.
```

**Other rules**

```
<comment> ::= '"' {<any-character>} '"'.
<edge> ::= 'edge' <edge_name> {<token>}+.
<node> ::= 'node' <node_name> <port_name> {<token>}+.
<token> ::= <string> <tokentype>.
<tokentype> ::= '<N>' | '<S>'.
<quote> ::= '"'.
<string> ::= <quote> {<any-character>} <quote>.
```

Node, edge and port names can consist of any character except the following characters: space ' ', tab '\t', carriage-return '\r' and eof-string '\0'.

The tokentype can either be a number "<N>" or a symbol "<S>".

9.3 Token administration during simulation

The third way tokens are placed and removed in a graph, is by execution of nodes. The execution of a node is done by the execution of a corresponding C function. Such an execution function has one single argument which is a pointer to the network representation of the node.

The default procedure to execute a node is to fetch the input tokens from the incoming edges and create one or more new tokens which are placed on the output edges.

Placing and removing tokens is done by the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_token_from_port</code></td>
<td>fetches a token from a port of a node</td>
</tr>
<tr>
<td><code>put_token_on_port</code></td>
<td>places a token on a port of a node</td>
</tr>
</tbody>
</table>

It should also be possible to destroy and create tokens. The functions for doing this are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>free_token</code></td>
<td>frees a token</td>
</tr>
<tr>
<td><code>alloc_token</code></td>
<td>Creates a new token</td>
</tr>
</tbody>
</table>

Another feature of the tokenflow simulator is the execution of nodes in either a symbolic or a numeric mode. This mode is determined by the types of the input tokens. If all input tokens have the same type, no conversion has to be made. But it is also possible that some tokens have different types. In this case, two options are possible. Either the tokens can be converted to the same type and all these tokens can be processed, or the tokens cannot be converted to the same type and in this case an error should be generated.

The type checking and conversion is done by the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_to_tokenlink</code></td>
<td>Places the tokenpointer in a linked list.</td>
</tr>
</tbody>
</table>
Function | convert.to_one.type (tokenlink) ()
---|---
Description | Returns the tokentype of the tokens in the tokenlist. If tokenlist contains numeric tokens as well as symbolic tokens, then these number tokens are converted to symbolic tokens.

Each time a token is taken from an edge, the function add_to_tokenlink must be called. This will place the token in a linked list. The tokens in the list can be converted to the same type by using the function convert.to_one.type. The LINK type is used for building the linked list of pointers to tokens.

How these functions are used is listed below.

```c
void dfgsim_add (TSIMNODE_PTR node) {

    TSIMTOKEN_PTR in1, in2, out;
    char newsym[SYMBOLSIZ];
    LINK tokenlink = NULL;

    /* [STEP 1] get input token(s) */
    in1 = get_token_from_port (node, 'N-0');
    add_to_tokenlink (&tokenlink, in1);
    in2 = get_token_from_port (node, 'N-1');
    add_to_tokenlink (&tokenlink, in2);

    /* [STEP 2] create output token(s) */
    out = alloc_token();
    out->type = convert_to_one_type (&tokenlink);
    switch (out->type) {
        case TSIM_TOKEN_NUMBER:
            out->value.number = in1->value.number + in2->value.number;
            break;
        case TSIM_TOKEN_SYMBOL:
            sprintf (newsym, '(+%s %s)', in1->value.symbol, in2->value.symbol);
            out->value.symbol = NEW_STRING (newsym);
            break;
    }
    put_token_on_port (node, 'N-2', out);

    /* [STEP 3] remove input tokens */
    free_token(in1);
    free_token(in2);

    /* [STEP 4] free token link structure */
    free_link(tokenlink);
}
```

The example shows an implementation of an add node. This node can execute both numeric or symbolic way.
Chapter 10

DFG and CTG simulation modes

In this chapter the DFG and CTG simulation modes are described. A DFG can either be simulated stand-alone or controlled by a CTG graph. These simulation modes are known as DFG mode and CTG mode respectively.

10.1 Simulation modes

The token flow simulator can operate in two modes:

**DFG** This is the DFG stand-alone simulation. Nodes which are to be executed in one single step are nodes which have sufficient tokens on their input ports for execution.

**CONTROLLED DFG** In this mode, the execution order of DFG nodes is determined by a control graph (CTG). A complete specification of the CTG node types can be found in appendix A.2. The most important CTG node type is the state node. A state node determines which DFG nodes are to be executed in one single CTG step.

The two different simulation modes are illustrated in figure 10.1. Tokens are represented by dots and the nodes which are to be executed in the next step, are marked by a square.

In CTG mode, so called node links determine which DFG nodes have to be executed, if a CTG node is executed. These *node links* are situated in the network data structure of the CTG node and are defined by the inter-relation specification of graphs, in which links between nodes of DFG, CTG and NWG graphs are specified.

It is also possible that a DFG node has to be executed during two or more CTG steps (multi cycles). Two or more CTG nodes will have node links to the same DFG node. The DFG node can only be executed once. So if the same DFG node has been executed a second time, it will simply be skipped.
10.2 DFG mode

10.2.1 Framework for DFG mode simulation

The execution of a step in the DFG mode is done by one single function call, which performs the following steps:

- Place all nodes of the CEL into the FEL.
- Evaluate all events in the CEL (execute nodes).
- If the FEL is empty, return False otherwise return True.

While executing a node, other nodes may become executable and are put in the FEL. Execution of the whole graph is done by repeatedly calling the single step function until it returns False. As a result no executable nodes will remain in the graph. If the graph is generated properly, the only tokens in the graph should be the tokens on the output nodes (except tokens on the control ports of entry nodes).

Before a node is selected from the CEL for simulation, it is checked a second time whether or not it can be simulated. This has to be done because it is possible that tokens are removed between two simulation steps by user interaction.

10.2.2 Implementation of DFG nodes

In the following part the implementation of the DFG nodes will be treated.
**Operation nodes**

The implementation of operation nodes is straightforward. First a token is taken from each incoming edge. A new token is created and according to the type of the input tokens, a new output token is created.

If not all input tokens have the same token type, numeric type tokens are converted to symbolic tokens. This allows a mixed-type simulation. If all input tokens are numeric, the node will also be executed numerically. Only if one or more tokens are symbolic, the node is executed symbolically.

**Input and Output nodes**

These nodes are not implemented because they have not to be executed.

**Constant node**

The input token is removed and an output token with the same token type as the input token is generated which contains the constant value.

This value of the constant is determined by the property named 'const' and can either be specified interactively or in the DFG textformat.

**Branch, merge, entry and exit nodes**

Merge and entry nodes have the same implementation. The token on the control input port is taken first and the value of this token determines from which input the token has to be passed to the output port.

Normally, nodes are executable if all input ports contain tokens. But merge and entry nodes are already executable as soon as the control port and the selected port contain a token.

If an entry node is reset, this node will get a zero value numeric token placed on the control port. This will initialize the loop which is constructed with the entry node.

Branch and exit nodes also have the same implementation. First of all the token from the control input is taken and the value of this token determines to which output port
the token taken from the input port, has to be passed.

Get and Put nodes

Get and put nodes are used to import and export tokens from and to an external physical port.

First of all a chain token is taken from the chain input port. In the case of a get node, a window pops up for entering a new token value. Momentary the type of the chain token determines the type of the new token, but it should be possible to let the user choose the type. In case of a put node a second token has to be taken but now from the data input port and is displayed in a view window. At the end, the chain token is passed to the chain output port.

Array operations

Initialization of an array is done by the execution of an array node. A chain token (used for chaining of array operations) is taken from the single input of the node and an one dimensional array is allocated for the storage of tokens. A pointer to this array is situated in the network data structure of the array node. The number of dimensions and the size of each dimension is determined by the node property 'array-dim', which is specified in the DFG textformat. Execution of the array node is terminated by the passing of the chain token to the chain output port.

It is also possible to initialize the array with tokens. This can be done by specifying the contents of the array by using the 'const-value' in the DFG textformat.

Updating the array is done by using the update node. When such a node is executed the chain input token is taken from the chain input, the index is fetched from the index ports and a data value is taken from the data input port. The actual updating of the array is done by calculation of the linear index in the one dimensional array of the array node. This calculation uses the maximum index values per dimension which are set when the array node is executed and are stored in the network data structure of the corresponding array node. Next, the value in the array at the calculated index is updated with the token taken from the data input port. The last thing which has to be done is the passing of the chain token to the chain output port.

Retrieving a token in the array is done by using the retrieve node. Chain handling and calculation of the index is done in the same way as during the execution of an
update node. The only difference is that a token is fetched from the array at the position indicated by the index and it is passed to the data output port.

The array, used by retrieve and update nodes, is situated at the array node data structure. To link the retrieve and update nodes to an array node, a pointer exists in the network data structures of these nodes, which points to the array node.

Other operations

The remaining nodes are the noop, nothing, construct, value and delay node. The implementation of these nodes is nothing more then passing a token is from the input to the output port.

10.3 CTG mode

For simulation of a DFG graph in CTG mode, the corresponding (linked) CTG has to be executed. Each time a CTG node is executed, the corresponding DFG nodes are evaluated. The links between CTG and DFG nodes are specified by node links. So the simulation of a DFG graph is controlled by its CTG graph. If a CTG graph is reset, the corresponding DFG graph will also be reset.

10.3.1 Framework for CTG mode simulation

Execution of a step in CTG mode is done by the function called \textit{ctg.step}. This function will perform the following steps after each other.

- Place all nodes of the CEL in the FEL of the CTG.
- Evaluate all events in the CEL of the CTG (execute CTG nodes). For each CTG node all linked DFG node are evaluated.
- If the FEL of the CTG is empty, return False else return True.

10.3.2 Implementation of CTG nodes

In the following part the implementation of the CTG nodes shall be treated. The specification of the CTG node types can be found in appendix A.2.
**DFG and CTG simulation nodes**

**State node**

First, a token is taken from the input port. Next, a DFG simulation step is performed. The DFG nodes which are linked to the state node are placed in the CEL of the DFG. This is done by creating a CEL in which all DFG nodes, linked to the CTG node, are placed.

It is not required that all nodes have to be executable the moment the DFG step is started. Some nodes will become executable, by executing others. At the end no events, except those which have been evaluated in an earlier CTG step, will remain in the CEL of the DFG. To fulfil this requirement, it is necessary that there are enough tokens in the DFG graph, to do so.

**Branch, merge, entry and exit nodes**

Branch, merge, entry and exit nodes can be treated the same way as their DFG variants. The only difference is that the CTG versions have no control ports. Each of these CTG nodes are linked to their DFG variants, which are evaluated by executing these CTG nodes. The control token of the DFG variant is also used in the CTG variant for selection purposes.

**Fork and join nodes**

Execution of a fork node is done by taking a token from the input port and place it on all output ports. A join node behaves like the inverse of a fork node. From each input port a token is taken and a new output token is created which has the same type as the input tokens. This new token is placed on the output port.

**Input and output nodes**

These nodes are not implemented because they have not to be executed.

**10.4 Simulation of subgraphs**

The simulation of subgraph is only implemented for DFGs and is done by using an *eventlist stack.*
The method for simulating a subgraph is the following:

- Push the current event list on the \textit{cel.dfg.stack}.
- Reset the subgraph.
- Get the tokens from all input ports of the subgraph node and place them on the outgoing edges of the corresponding input nodes of the subgraph.
- By placing these tokens in the subgraph the future event list will be filled with events. So the subgraph is executed by repeatedly execution of the \texttt{dfg.step} function until it returns \texttt{False}.
- Restore the current event list by a pop operation on the \textit{cel.dfg.stack}.
- Take all output tokens of the subgraph and place them on the output ports of the subgraph node.

By using a stack mechanism, nestings of subgraphs can be simulated properly.

\section*{10.5 Mixed-level simulation of the \textit{tokenflow} simulator with ESCAPE's discrete event simulator}

In ESCAPE modules are used to design circuits. The behaviour of a module can be specified by using a High-level Description Language (HDL). Besides specification of a module in a HDL, it is also possible to specify the behaviour of a module by a DFG graph. This behaviour specification level for modules is currently still under development.

The behaviour of a module, specified by a DFG, is evaluated whenever the \textit{start} signal becomes high. Each net value is transformed into a token and placed on the output port of the corresponding input node. The DFG simulator is called and after a specified delay time the \textit{ready} signal will become high. The resulting output tokens of the DFG are now transformed into net values.

This simulation model is illustrated in figure 10.2.

The main advantage of this module specification level is that it is easier to map a HDL on a DFG graph and to simulate this graph, than to implement a simulator for this HDL. The behaviour of the DFG will be the same as the behaviour specified by the HDL specification. So afterwards, the results of the DFG simulation can be mapped on e.g. the corresponding variables used in the HDL specification.
Figure 10.2: Module behaviour specified by a DFG
Chapter 11

Rewriting symbolic token expressions

During symbolic graph simulation, token expressions can get very complex and the need arises to simplify them during simulation or afterwards. In this chapter, it is described how symbolic tokens can be rewritten using a term-rewriting system.

11.1 Term-rewriting systems

A program called REWRITE is being developed, which can be used to rewrite symbolic expressions according to a set of rewrite rules. It is used to simplify the expressions of symbolic tokens by defining the appropriate rules.

Normal form of an expression

Expressions can be rewritten by making them equivalent to the lefthand side of a rewrite rule by replacing the variables in that rewrite rule by expressions. If no sub-expressions of an expression can be rewritten, this expression is a normal form.

Term-rewriting systems must satisfy two important properties:

Strongly terminating No expressions exist which can be rewritten infinitely many times.

Confluence Every expression has a unique normal form.

A possible application area of the rewriting system could be to prove the functional equivalency of two or more graphs. Also it could be possible to simplify graphs which are used as subgraphs in a larger context: due to the context in which the subgraph is executed, it might be possible to simplify the graph.
11.2 Interface between ESCAPE and REWRITE

The communication between ESCAPE and the external program REWRITE is done by using the intertool protocol (ITP) of ESCAPE (figure 11.1). In this protocol, instructions can be sent and received using ASCII keyword commands.

![Keyword commands](image)

Figure 11.1: Interface between ESCAPE and REWRITE

The following keyword commands are used for communication from ESCAPE to REWRITE:

<table>
<thead>
<tr>
<th>Keyword command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-cons &lt;format&gt;</td>
<td>Sets the constant format.</td>
</tr>
<tr>
<td></td>
<td>Momentary format: Nat32</td>
</tr>
<tr>
<td>set-mode user/lisp</td>
<td>Sets the return format, which can either be user friendly or lisp.</td>
</tr>
<tr>
<td>rewrite &lt;expression&gt;</td>
<td>For rewriting an expression.</td>
</tr>
<tr>
<td>quit</td>
<td>To quit the REWRITE application.</td>
</tr>
</tbody>
</table>

The following keyword commands are used for communication from REWRITE to ESCAPE:

<table>
<thead>
<tr>
<th>Keyword command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression user/lisp</td>
<td>To return the rewritten expression.</td>
</tr>
<tr>
<td>&lt;expression&gt;</td>
<td>The format of the expression can either be user friendly or lisp.</td>
</tr>
<tr>
<td>error</td>
<td>Used to indicated that an error has occurred.</td>
</tr>
</tbody>
</table>

Before an expression can be rewritten, the constant format has to be set (e.g. Nat32, 32 bit integer). Also the return format of the expression has to be set.

The communication is asynchronous. This implies that ESCAPE has to administrate the sequence of tokens transmitted by REWRITE and match each received rewritten expression with its original.

11.3 User interface for rewriting tokens

An user interface has been provided for selecting and rewriting tokens on edges. Also is it possible to replace the token with its rewritten version.
11.4 Possible future extensions

The development of the rewrite interface is still in the prototype phase. Sometimes it is desirable that some groups of rewrite rules are turned off by the user. Also the possibility must exist to substitute variables by either numbers or expressions. Perhaps it is also useful to provide a feature for automatic rewriting of symbolic tokens during simulation.

Another interesting future extension might be the complete symbolic simulation of branch-merge and entry-exit combinations. E.g. the token on the control port of a branch isn't used for selecting the output port, but is used to create an if expression which is passed onto each output port of the branch node.
Conclusions and Future work

In order to handle different graph models, ESCAPE has been extended with a graph view. The definitions of DFGs and related graphs have been developed to support their usage in ESCAPE. For simulation of DFGs, a token flow simulator has been developed. The current status of the graph view and the token flow simulator, is listed below:

Graph view - Interactive editing of graphs

- The graphical description and the behaviour of graphs can be defined by a textformat.
- Reading of a graph textformat.
- Interactive editing of graphs.
- Saving and loading a graph in the ESCAPE file format.

Token flow simulator - Simulation of graphs

- DFG and CTG numeric simulation with data values limited to 32 bit.
- Symbolic simulation of a number of DFG and CTG nodes. (e.g. operation, constant, get, put and array type nodes).
- A mixed-mode simulation of symbolic and numeric tokens.
- Simulation of DFG subgraphs.
- Mixed-level simulation of the tokenflow simulator with ESCAPE's discrete event simulator (still under development).
A whole range of extensions is could be thought of. I think these future extensions will mainly depend on the application area of the tokenflow simulator. Application areas for the tokenflow simulation could be behaviour verification, education or a mixed-level simulation with other simulator types. An other application could be the execution of all kind of algorithms on graphs (e.g. a depth first search algorithm).

The current form of the simulator is such that is only intended for DFG and CTG graphs.

I think it will be useful to make the simulator operate on different types of tokenflow graph models. In this way the application area of the simulator will be enlarged. E.g. petri net simulation might be possible in the future.

I think the main goal of this simulator will be to create a better understanding of DFGs and to simulate various HDL types by using the DFG standard as an intermediate format for simulation. In this way a whole range of HDLs can be simulated without much effort.
Appendix

Graph definitions

In this appendix the specification of the nodes and edges of the data flow graph (DFG), control flow graph (CTG) and network graph (NWG) are given.

A.1 DFG graph

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>numeric addition</td>
<td>'N-0' and 'N-1'</td>
<td>'N-2'</td>
</tr>
<tr>
<td>*</td>
<td>numeric multiplication</td>
<td>'N-0' and 'N-1'</td>
<td>'N-2'</td>
</tr>
<tr>
<td>/</td>
<td>numeric divide</td>
<td>'left' and 'right'</td>
<td>'N-2'</td>
</tr>
<tr>
<td>%</td>
<td>numeric modulo</td>
<td>'left' and 'right'</td>
<td>'N-2'</td>
</tr>
<tr>
<td>-</td>
<td>numeric subtract</td>
<td>'left' and 'right'</td>
<td>'N-2'</td>
</tr>
<tr>
<td>++</td>
<td>numeric increment</td>
<td>'N-0'</td>
<td>'N-1'</td>
</tr>
<tr>
<td>-</td>
<td>numeric decrement</td>
<td>'N-0'</td>
<td>'N-1'</td>
</tr>
<tr>
<td>==</td>
<td>compares inputs</td>
<td>'N-0' and 'N-1' (numeric/bitvector)</td>
<td>'N-2' (boolean)</td>
</tr>
<tr>
<td>!=</td>
<td>compares inputs</td>
<td>'N-0' and 'N-1' (numeric/bitvector)</td>
<td>'N-2' (boolean)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>compares inputs</td>
<td>'left' and 'right' (numeric/bitvector)</td>
<td>'N-2' (boolean)</td>
</tr>
<tr>
<td>&gt;</td>
<td>compares inputs</td>
<td>'left' and 'right' (numeric/bitvector)</td>
<td>'N-2' (boolean)</td>
</tr>
<tr>
<td>&lt;</td>
<td>compares inputs</td>
<td>'left' and 'right' (numeric/bitvector)</td>
<td>'N-2' (boolean)</td>
</tr>
<tr>
<td>const</td>
<td>constant</td>
<td>'N-0'</td>
<td>'N-1'</td>
</tr>
</tbody>
</table>

Predefined node types

Arithmetic nodes

Numeric operations

Default inputs- and output types: numeric.

Bitwise operations

Default inputs- and output types: bitvectors.
Graphical simulation of Data Flow Graphs

Predefined edge types

data

Default type for edges. A data edge can have a data type and a width attached to it. The following data types are provided: The unsigned, signed magnitude and two complement integer types, Fixed point type and boolean type.

sequence

Sequence edges are used to enforce a required execution ordering on the nodes in the DFG.

Control nodes

Input- and output nodes

Array operations
A.2 Control graph

Predefined node types

State nodes

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>state node</td>
<td>'in'</td>
<td>'out'</td>
</tr>
</tbody>
</table>

Control nodes

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>to build 'if-then-else' constructs</td>
<td>'in'</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>merge</td>
<td>to build 'if-then-else' constructs</td>
<td>'in'</td>
<td>'out'</td>
</tr>
<tr>
<td>entry</td>
<td>to build 'loop' constructs</td>
<td>'in'</td>
<td>'out'</td>
</tr>
<tr>
<td>exit</td>
<td>to build 'loop' constructs</td>
<td>'in'</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>fork</td>
<td>to start concurrency</td>
<td>'in'</td>
<td>undetermined number (&gt; 1)</td>
</tr>
<tr>
<td>join</td>
<td>to merge concurrency</td>
<td>undetermined number (&gt; 1)</td>
<td>'out'</td>
</tr>
</tbody>
</table>

Input- and outputnodes

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input interface node for any graph</td>
<td>no input ports</td>
<td>'out'</td>
</tr>
<tr>
<td>output</td>
<td>output interface node for any graph</td>
<td>'in'</td>
<td>no output ports</td>
</tr>
</tbody>
</table>

Predefined edge types

sequence

Default type for edges.

A.3 Network graph

Node types

Operation nodes

Description: Instantiations of library modules and other created modules.

Interface:

- undetermined number of (input/output/in-out) ports for data
- optional input port for control
- optional input port for clock
Graphical simulation of Data Flow Graphs

- optional input port for start-signal (1 bit)
- optional output port for ready-signal (1 bit)
- optional output port for flags

**Multiplex nodes**

Description: All kinds of multiplexers

Interface:

- undetermined number of input ports for data
- one input port for control
- one output port for data

**DeMultiplex nodes**

Description: All kinds of demultiplexers

Interface:

- one input port for data
- one input port for control
- undetermined number of output ports for data

**TriState nodes**

Description: Tri-state buffers

Interface:

- one input port for data
- one input port for control (1 bit)
- one output port for data

**Predefined node types**

**Register nodes**

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Edge triggered register</td>
<td>one date and one clock input and one data output</td>
</tr>
</tbody>
</table>
Graph definitions

Bus nodes

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus</td>
<td>Connects modules to each other</td>
<td>undetermined number of (input/output/in-out) ports</td>
</tr>
</tbody>
</table>

Input- and outputnodes

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input interface node for any graph</td>
<td>one output port</td>
</tr>
<tr>
<td>output</td>
<td>output interface node for any graph</td>
<td>one input port</td>
</tr>
<tr>
<td>in-out</td>
<td>io interface node for any graph</td>
<td>one in-out port</td>
</tr>
</tbody>
</table>

Input- and outputpads

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
<th>ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-pad</td>
<td>input interface node for a design</td>
<td>one output port</td>
</tr>
<tr>
<td>out-pad</td>
<td>output interface node for a design</td>
<td>one input port</td>
</tr>
<tr>
<td>io-pad</td>
<td>io interface node for a design</td>
<td>one in-out port</td>
</tr>
</tbody>
</table>

Predefined edge types

data

A network edge can only be of type data and has a 'width' attached to it. It can also have a 'connection' to specify how bits of ports are connected. (It always connects a module to a bus-node.)
Appendix B

GDEF Specification

In this appendix the specification of the GDEF language can be found.

The basic form of the syntax is very simple: each object is represented as a list. These lists start with an opening brace followed by a keyword. Items following the keyword can be strings, integers or other objects. An object is closed by a closing brace.
GDEF Specification

```
gdef format:

swtiches:

grid:

directed:

use_pins:
```
Graphical simulation of Data Flow Graphs

node:

edge:

node-descr:

fillcolor:

boundobj:

symbols:
ports:

menu:

function:

props:

symbol:

circle:


circle:

Graphical simulation of Data Flow Graphs

- **line:**
  - ( line integer integer integer integer )

- **poly:**
  - ( poly point )

- **text:**
  - ( text string integer integer align )

- **points:**
  - ( integer integer )

- **align:**
  - centered
    - left_aligned
    - right_aligned

- **portdef:**
  - in-port
  - out-port
  - inout-port
in-port:

\[
\begin{array}{c}
\text{input} \rightarrow \text{string} \rightarrow \text{portitem} \\
\text{portitem} \rightarrow \text{string} \rightarrow \text{output}
\end{array}
\]

out-port:

\[
\begin{array}{c}
\text{output} \rightarrow \text{string} \rightarrow \text{portitem} \\
\text{portitem} \rightarrow \text{string} \rightarrow \text{output}
\end{array}
\]

inout-port:

\[
\begin{array}{c}
\text{inout} \rightarrow \text{string} \rightarrow \text{portitem} \\
\text{portitem} \rightarrow \text{string} \rightarrow \text{inout}
\end{array}
\]

portitem:

\[
\begin{array}{c}
\text{portpos} \\
\text{portpos-descr} \\
\text{portprop}
\end{array}
\]

portpos:

\[
\begin{array}{c}
\text{position} \rightarrow \text{portpos-descr} \\
\text{portpos-descr} \rightarrow \text{position}
\end{array}
\]

portprop:

\[
\begin{array}{c}
\text{property} \rightarrow \text{string} \\
\text{string} \rightarrow \text{property}
\end{array}
\]

portpos-descr:

\[
\begin{array}{c}
\text{pp-at-boundary} \\
\text{pp-fixed} \\
\text{pp-line} \\
\text{pp-circle}
\end{array}
\]

pp-at-boundary:

\[
\begin{array}{c}
\text{at-boundary}
\end{array}
\]
Graphical simulation of Data Flow Graphs

- **pp-fixed**:  
  - fixed -> integer -> integer

- **pp-line**:  
  - line -> integer -> integer -> integer -> integer

- **pp-circle**:  
  - circle -> ciritem

- **ciritem**:  
  - cir-radius
  - cir-origin
  - cir-from
  - cir-to

- **cir-radius**:  
  - (radius -> integer) ->

- **cir-origin**:  
  - (origin -> integer -> integer) ->

- **cir-from**:  
  - (from -> integer) ->

- **cir-to**:  
  - (to -> integer) ->
edge-descr:

```
edge-linestyle
  edge-color
  edge-props
```

edge-linestyle:

```
(line-style) linestyle ()
```

decode-color:

```
(color) string ()
```

decode-props:

```
(props) gprop ()
```

genstyle:

```
solid

dotted

dashed
```

gprop:

```
(prop) string (default) string ()
```
Graphical simulation of Data Flow Graphs

string:

integer:
The following rules have been added to the graph parser so it can read link graphs as well. The C macros which are used as building block for the construction of the rules are described in chapter 6.

```c
/*** link-view rule(s) ***/
Rule F_link_view Is Form( K_link_view, '4')
  Code
    view = LINK;
  NoOrder Flag( F_graph_link(), 0)
EndForm

Rule F_graph_link Is Form( K_graph_link, '6664')
  InOrder is_identifier()
  Code
    graph = CALLOC_STRUCT(DFG_GRAPH);
    graph->view = view;
    hash_add( graph_hash, graph->name, graph);
    node_hash = graph->node_hash hash_create( "node_hash",307,0);
    edge_hash = graph->edge_hash = hash_create( "edge hash",307,0);
    linknode = 1;
  NoOrder Flag( F_data_flow_name(), 0) ||
    Flag( F_control_name(), 1) ||
    Flag( F_network_name(), 2) ||
    Flag( F_node_link(), 3)
EndForm

Rule F_data_flow_name Is Form0( K_data_flow)
  InOrder is_identifier()
  Code
    graph->data_flow_name = new_string( dfg_lex_word);
EndForm0

Rule F_control_name Is Form0( K_control)
  InOrder is_identifier()
  Code
    graph->control_name = new_string( dfg_lex_word);
EndForm0

Rule F_network_name Is Form0( K_network)
  InOrder is_identifier()
  Code
    graph->network_name = new_string( dfg_lex_word);
EndForm0
```

73
Rule F_node_link Is Form( K_node_link, '2222')
    Code ( char unique[BUFSIZE];
        sprintf (unique, 'link%d', linknode);
        linknode++;
        if (!hash_get( node_hash, &unique, &node))
            node = new_node( node_hash, &unique);
        else if (node->type)
            parse_err( 'Node name '%s' not unique!'n, node->name);
    )
    NoOrder Flag( F_type_node(), 0 ) ||
    Flag( F_data_flow(), 1 ) ||
    Flag( F_control(), 2 ) ||
    Flag( F_network(), 3 )
EndForm

Rule F_data_flow Is Form( K_data_flow, '4')
    NoOrder Flag( is_identifier(), 0 )
    Code { NODENAME_LIST *new = CALLOC_STRUCT( NODENAME_LIST);
        new->name = new_string( dfg_lex_word);
        new->next = node->data_flow_list;
        node->data_flow_list = new;
    } EndForm

Rule F_control Is Form( K_control, '4')
    NoOrder Flag( is_identifier(), 0 )
    Code { NODENAME_LIST *new = CALLOC_STRUCT( NODENAME_LIST);
        new->name = new_string( dfg_lex_word);
        new->next = node->control_list;
        node->control_list = new;
    } EndForm

Rule F_network Is Form( K_network, '4')
    NoOrder Flag( is_identifier(), 0 )
    Code { NODENAME_LIST *new = CALLOC_STRUCT( NODENAME_LIST);
        new->name = new_string( dfg_lex_word);
        new->next = node->network_list;
        node->network_list = new;
    } EndForm
Bibliography

J.T.J. van Eijndhoven, G.G. de Jong and L. Stok
EUT Report 91-E-251, June 1991

[Fis73] Concepts and methods in discrete event digital simulation
Fishman G.S.
New York: John Wiley and Sons, 1973

J.W.G. Fleurkens
To be published

[Jia90] A novel asynchronous control of discrete-event simulation
Jianchao Z., Guoji S. and Baosheng H.
SCSC, Summer Computer Simulation Conference, Calgary, Atla., Canada, 16-18
july 1990, p7-11

[Hoo79] Simulation strategies: A theoretical and practical investigation.
Hooper J.W.

[Hoo82] An algorithmic analysis of simulation strategies
Hooper J.W. and Reilly K.D.

[Hoo86] Strategy related characteristics of discrete-event languages and models
Hooper J.W.
Simulation 46:4, 1986, p153-159

[New88] An interface description language for graph editors
Newbery F.J.
1988
[Rowe87] **A Browser for Directed Graphs** L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis and A. Tuan
Software - Practice and experience. vol. 17(1), 61-76, January 1987

[Sme88] **Simulatie en implementatie**
Smedinga R.
Amsterdam: Addison Wesley, 1988

[Shu78] **On the relative merits of two major methodologies for simulation model construction**
Shub C.M.

[Sug81] **Methods for Visual Understanding of Hierarchical System Structures.**
Sugiyama K.