Proc. of the Eur. Conf. on Design Automation (EDAC),
Brussles, Belgium, March 1992, pp. 193–199

A Data Flow Graph Exchange Standard*

Jos T.J. van Eijndhoven Leon Stok

Eindhoven University of Technology
The Netherlands

Abstract

The paper presents a data flow graph exchange standard, agreed upon and used by the partners in the ESPRIT research project ASCIS. These data flow graphs are generated from known user interface languages such as Silage, VHDL, and C, and are used to drive architectural synthesis packages and formal verification. The graph semantics are defined to offer a unique degree of freedom for time and area optimizations in synthesis, by giving a maximal parallel representation and combining control and data flow in a consistent way. The graph textual exchange format was developed to allow site and application dependent extensions, without disturbing tools who do not know about these. This way both upwards and downwards compatibility is obtained for future extensions. A very broad application area is intended, covering control dominated designs, highly and lowly multiplexed data paths, up to systolic arrays.

1. Introduction

This paper describes a data flow graph (DFG) standard for the synthesis and verification of integrated circuits from a behavioural level description. Its development started back in 1987. In 1990 the format and its semantics were adopted by the ASCIS project (ESPRIT Basic Research Action 3281), a cooperation between seven universities and research institutes throughout Europe. The format is intended as intermediate form between user oriented interfaces (languages, schematics) and synthesis and verification tools, as well as serving as interchange format between different machines or sites. Therefor the accurate definition of the graph semantics are of utmost importance. Allowing maximal freedom to synthesis tools for the generation of different solutions in different architectural styles without imposing any unnecessary restriction, was another development goal. Due to the long term research aspect of the related work and the different requirements of individual sites and tools, flexibility and extensibility were major design goals for the file format. A textual format was strongly preferred over a binary format for several reasons, but human readability was hardly a design issue. The intended usage is illustrated in Figure 1.

The advantage of this scheme is of course the separation of the different synthesis tools from the complex user level interfaces. The DFG format is a relatively simple interface, both in syntax and in semantics. The detailed semantical interpretation of languages like VHDL and ELLA for synthesis purposes is not trivial.

Due to the EDIF and LISP like structure of the format with braces and keywords, individual tools and sites can add information to it, without disturbing other tools who do not know about these. Thus the format can optionally be used throughout the synthesis process, by repeatedly annotating

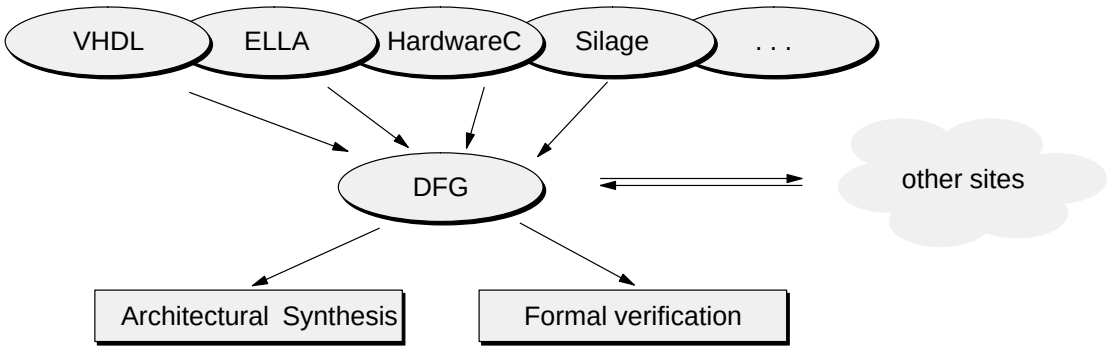

Figure 1. DFG positioning

* This research is sponsored in part by the European Community under contract ESPRIT BRA 3281

the graph with the results of individual tools. This leads to a schema as in Figure 2.

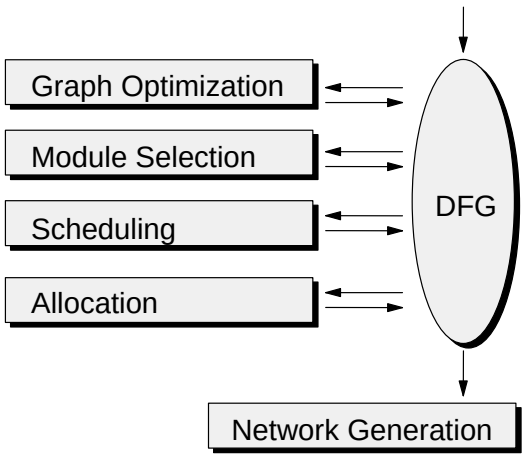

Figure 2. Extended use of the DFG.

The decoupling of the synthesis software in several smaller tools, greatly improves the manageability of such a system, and allows a choice of algorithms for the basic tasks. On modern workstations the parsing and generation of the DFG text files is extremely fast, and hence not of real concern.

The employed graph model does not distinguish between data path and control flow, and allows cycles to model loops in the algorithmic behavior, providing maximal freedom for different implementation styles. The uniform and combined treatment of data and control has resulted in both a concise semantical definition and an extreme flexibility for architectural synthesis. The DFG provides a maximal parallel description of the algorithm, from which many design alternatives can be generated. The generation of such a graph from a procedural sequential programming language requires a full data flow analysis, involving a detailed lifetime and scope analysis, through conditional statements, loops, and procedure interfaces. The complexity of this task is n×n with n the number of operations in the input text. In practice this means that descriptions of relatively complex chips (programs of several hundred lines) can be converted to a DFG representation in a few seconds cpu time on a standard workstation. The combination of the consistent merging of data and control flow even for loops, and the maximal parallel representation are unique features of this model. It is also the data flow analysis that resolves the very different nature of the behavioural input languages into a uniform semantics. Hence the difficulty of this step depends upon the specific language. In particular full VHDL is very hard due to its event driven semantics, and subsetting is almost inevitable as in other synthesis systems.

2. Related work

The use of data flow graphs or demand graphs is well accepted in architectural synthesis. However closer examination shows a wide variety in the concepts used.

2.1 Tree representations

Some systems use internally a tree representation instead of a real data flow graph. This tree is often similar to a parse tree, from a supported textual interface [Hilfin84, Marwed85]. However such a representation is not well suited to support many algorithms in synthesis, such as structural optimizations. Hence implementations become unnecessary complex, must create derived data structures, or just overlook optimization possibilities.

2.2 Semi data flow representations

Several systems do not have a real data flow graph where the edges represent *values*, but stick to operations and *variables* instead. These variables can get modelled with nodes [Knapp85, Campos89]. Such a representation is easily built from procedural programming languages, since no real data flow analysis is required. If this analysis is not done, the representation contains many timing and mapping restrictions which are not necessary, and severely limits the search space for optimal solutions of the generated architecture. A related approach is the introduction of sequence edges to denote the sequential ordering of operations as found in a (procedural) input language [Brayton88].

2.3 Separation of data and control flow

Many systems limit the data flow analysis to so-called 'straight line code' only, generating several blocks with data flow code [Lis88, Pangrle87]. Conditional constructs, loops, and procedure calls are represented in a separate control graph. Although relatively easy to implement, this again severely limits many algorithms (graph optimizations, scheduling, allocation) in their search space, increasing the risk of staying with suboptimal solutions. To overcome the most severe limitations, simple conditional constructs might get moved into the data flow blocks [Walker87][Paulin91]. The imposed limitation of having clock cycle boundaries around these blocks (at least for loop bodies) is silently accepted.

2.4 Our exchange format

Since individual synthesis systems adopt different approaches and limitations, an exchange format designed to bridge such systems should by itself not impose comparable restrictions, to allow a fair design exchange. The freedom to allow a clock cycle boundary anywhere in a loop, instead of being forced to put this at the loop start or loop end, or even to choose this boundary at different places for individual

variables, seems a unique generality offered by our approach. The supported full data flow analysis through loop and procedure call interfaces, furthermore turns often required tasks like loop unrolling or procedure inline expansion into trivial operations. Conditional and loop statements in the input lead to a comparable block structuring of our graphs, allowing efficient synthesis algorithms [Stok91]. Another combined data and control flow graph was developed in the SPRITE project, which seems more confined towards signal processing applications, due to limited communication functionality [Krol92].

3. The Data Flow Graph

3.1 The basic approach

The DFG consists of nodes and directed edges. The *nodes* represent *operations* in the behavioural specification, and the *edges* model the transfer of *values* between these. Thus a single edge indicates that the result of one operation is passed to the argument of another one. One single data value instance is defined to be a *token*. We define the *execution* of an operation (node) as the process where a token is fetched and removed from the incoming edges, and a token containing the result of the operation is put on each outgoing edge. The execution order of the operations in the graph is thus constrained by the partial ordering of the nodes as defined by the directed edges.

Every argument of an operation can be seen as an *input port* of the operation, and the single result can be seen as the *output port* of the operation. It is required for a DFG that only one incoming edge is allowed to arrive on any input port. However several outgoing edges can leave from the output port. In general nodes can have several output ports, each with zero or more outgoing edges. However in many cases (commutative operations with one single output) there is no reason to distinguish between individual ports on a node, and hence ports are often left undefined.

In theory we will allow multiple tokens to be resident (queued) on any edge, giving maximal freedom in scheduling the execution of operations over time. However everybody who wants to use (create) such a schedule is itself responsible for synthesizing suitable hardware for these queues. Due to the DFG properties, the finally resulting values (tokens) do not depend upon the chosen execution order, and it is always possible to choose a restricted order of evaluation in which never more than one token is resident on any edge [Jong91].

This token passing method, defines the algorithmic behavior of the graph, without imposing any restrictions on the timing of the hardware to be generated. This was regarded a very important property. By adding special 'timing' edges to the graph, constraints on the time behavior can be specified if so desired. Furthermore the synthesis system is still free to choose any clocking scheme.

To provide a more accurate definition of this execution behavior, we will distinguish between a few different *node types*.

3.2 Operation nodes

Operations can be arithmetic, like $\times$, –, +, ++, or boolean like $\wedge$, $\vee$, $<$, $\equiv$, or can be more complex functions. The available set of operations is not defined nor restricted by the DFG format. However to successfully exchange designs between sites, it is required to agree on the names of a few basic operations, their allowed number of inputs, and –especially for non–commutative operations– on the names of their input ports. The format also provides for nesting of graphs in the same way as procedures in normal programming languages. The instantiation of another graph is performed by using a node type, referring to the name of a graph defined elsewhere in the textual description. Despite this node type, an instantiation distinguishes itself in no way from a normal basic operation, and hence these can be regarded as instantiations of implicitly defined procedures. The execution of an operation thus might involve the execution of a complex subgraph.

3.3 Input and Output nodes

Every graph requires at least one node of type input, and one of type output. Nodes of type input are the **only** nodes without input ports, and nodes of type output are the **only** nodes without output ports. If the graph would be instantiated elsewhere as operation, then the names of these input and output nodes define the port names of the operation.

For more complex input/output communication the put and get nodes should be used as defined later.

3.4 Constant nodes

Nodes of type 'constant' are nodes which generate a constant data value at their single output port. To indicate when this token is to be produced, these nodes also have one input port, and hence can be treated in the same way as unary operators. Edges with the sole purpose to activate 'constant' nodes are distinguished by giving them type 'source', to indicate that the data value of their tokens is actually ignored. The usage of a constant node, together with a simple operation and a pair of input/output nodes is depicted in Figure 3.

3.5 Branch and Merge nodes

A branch node passes the token from the incoming data edge to one output port, which is identified (selected) by the value of the token on the control input. Thus the node can be executed if both inputs have a token, and as result one token will appear on precisely one output port.

Merge nodes are dual to branch nodes passing a token from just one incoming data edge, selected by the value of the token on the control edge, to the single output port. (The

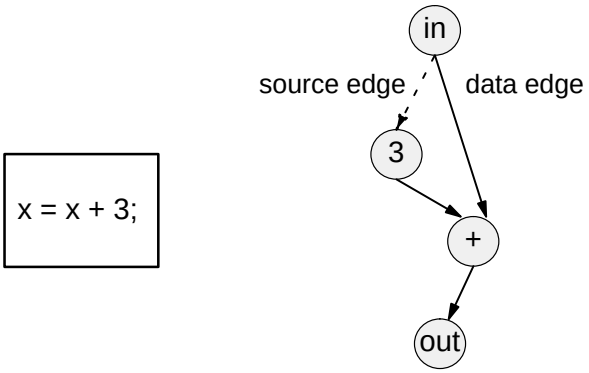

Figure 3. Simple DFG with constant node.

execution rule of this node is different: it can execute as soon as a token is available on the control input as well as on the selected input port.)

The branch and merge nodes are necessary for algorithmic constructs like if ... then ... else, or case ... of. An example of such a construct is shown in Figure 4.

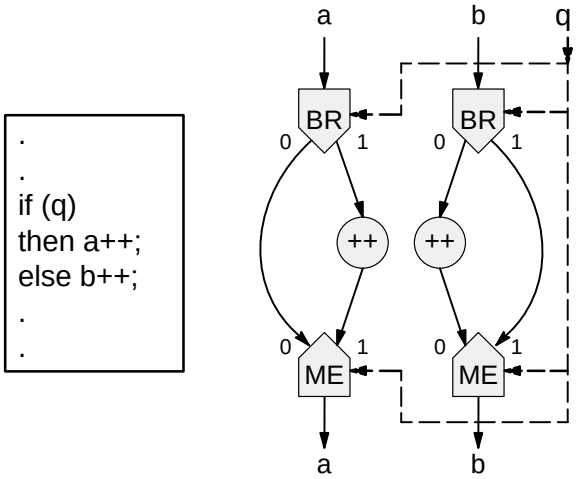

Figure 4. DFG if–then–else example.

3.6 Exit and Entry nodes

Exit and entry nodes are functionally identical to the branch and merge nodes respectively. However these nodes are used to build loop constructs, which could originate from while ... do or for ... do statements. The connectivity of the exit and entry nodes to create a loop construct is depicted in Figure 5.

The loop construct introduces cycles in the DFG. This are cycles through entry, exit and loop body, and cycles through entry and loop test. These cycles are relatively easy to find, due to the structure of the loop construct, and the entry and exit nodes separate disjunct acyclic subgraphs from the environment. Other cycles are not allowed in the DFG. Note however that loop constructs can nest.

To allow a proper execution of these loops with the token passing method, a special initialization is required: When the execution of a graph is started, all entry nodes must

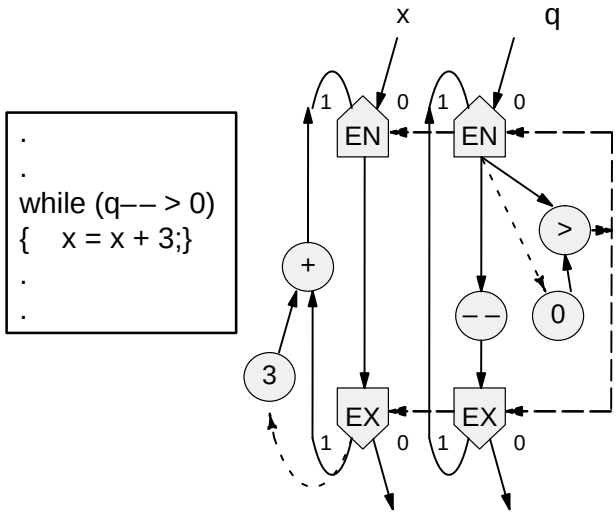

Figure 5. DFG while–do example.

obtain a token at their control input, selecting the input port for external data to enter the loop. The exit nodes do **not** obtain such an initialization token! 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.

Although the token passing semantics would give the impression of a sequential execution of the loop, this forms in no way a restriction towards different implementations. Note that the unrolling (unfolding) of a loop in the DFG is a simple and straightforward operation. If the loop is to be implemented sequentially, the token flow leaves maximum flexibility with respect to synchronization and timing issues. Of course clock cycle boundaries and state transitions must be introduced to execute the loop, but there is a free choice for the placement of this clock boundary: **it does not have to be at the entry nor at the exit nodes**. It is even perfectly allowed to desynchronize the loop cycling of different variables, for instance one variable might have completed 10 cycles, whereas another variable of the same loop has done just 2 cycles. However such asynchronous execution requires queueing of multiple tokens on (at least) the control edges.

3.7 Get and Put nodes

Get and put nodes are to provide a mechanism for communication protocols with the outside world and can appear anywhere in the graph, such as inside 'if' or 'loop' constructs, in contrast to 'input' and 'output' nodes. These get and put nodes make a reference to 'ports' through which external communication is to take place, which can for instance model pads on the chip boundary.

Get and put nodes which use one physical port are linked in sequential 'chains' to set the order in which read and write operations should appear on the port. When put and get nodes appear within 'if' or 'while' constructs, there chain

edges also appear in these constructs, with their own branch–merge or entry–exit node pairs. If put or get nodes for one physical port appear both in a main graph and in instantiated procedures, these edges should go through the procedure call and body by hereto added input and output ports on the procedure interface. The usage of put and get is depicted in Figure 6.

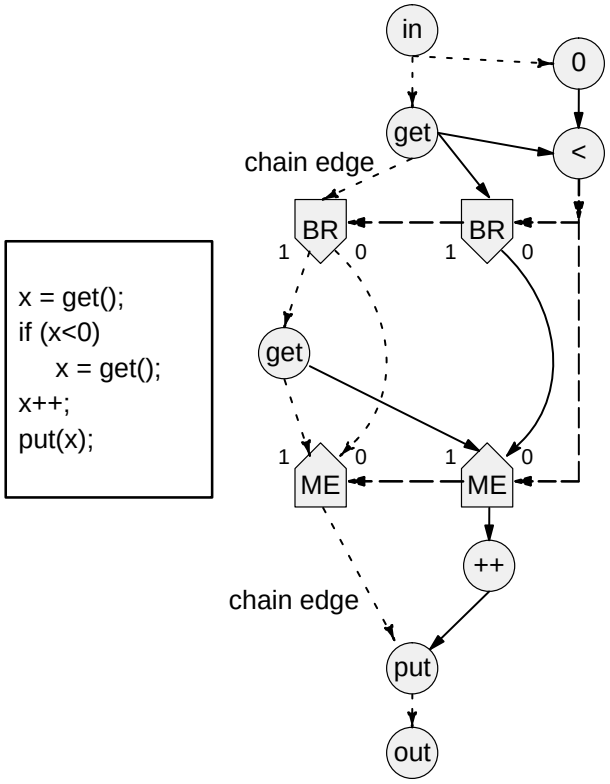

Figure 6. DFG get and put example.

3.8 Array operations

For operations on arrays three node types are provided:

The node type 'array' functions as array declaration. The declaration contains statements giving the dimensionality and size(s) of the array. Optionally initial values are provided. The 'array' node has outgoing chain edges, providing linkage to the other two related node types: 'retrieve' and 'update'.

The 'retrieve' nodes are used to read data values from the array. They have input ports for indices to address one value, and a data output port.The 'update' nodes are used to write values in the array. They have edges providing indices as the 'retrieve' nodes, and have a 'data' input port for the value to be written.

Chain edges are used to specify a (partial) ordering in which the retrieve and update operations must take place. Coming from a sequential input language, it seems natural to connect all retrieve and update nodes in one serial list (chain). However a careful analysis might reveal a degree of parallelism (independence), allowing more freedom in scheduling. For this purpose the chaining of retrieve and update nodes is in general structured as a connected acyclic graph. An example of array usage is given in Figure 7.

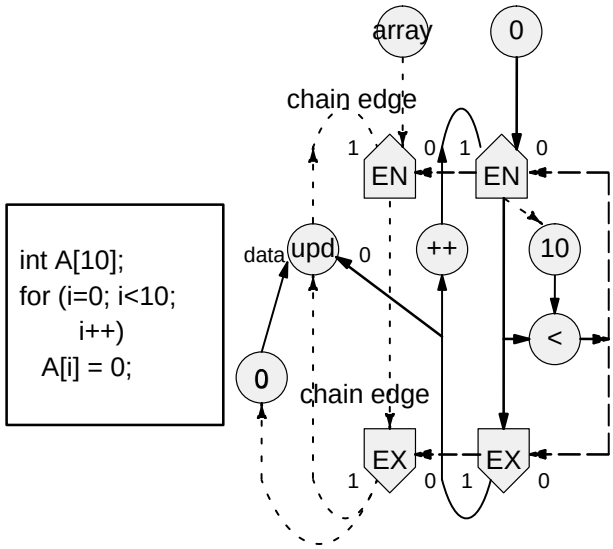

Figure 7. DFG array example

4. Data types and widths

4.1 Introduction

In the DFG data transport is represented by edges. Hence data types and data widths are (optional) properties of edges in the graph. The *data width* (edge width) is the number of bits that conceptually make up the data value and are involved in the data transfer. In general this can be different from the number of wires in the resulting hardware: The mapping to hardware could generate things as dual rail or bit serial transmission.

The *data type* determines the interpretation of the bit pattern to a numeric value. This data type is of importance for the selection of actual hardware modules performing the arithmetic operations of the nodes. The mechanism of selecting a hardware module depending on the data types of the attached edges can be referred to as *overloading.* The data type is furthermore required for the process of *width adjusting*. This is required when the width of an edge does not agree with the width of the port to which it connects.

4.2 Data type

A data type is attached to an edge with a *data type name*. Such a data type name is introduced and defined with file scope. This *data type def* statement is located outside any graph, introduces the data type name, and optionally specifies the numeric interpretation of the bit pattern. For now the syntax allows to express two's complement, unsigned, and signed magnitude, binary coded integer and fixed point numbers, as well as booleans.

Up to now no formats are defined to handle things like floating point numbers, text, records, or attributes as 'address of', as well as counting schemes which differ from the normally binary number representation such as bcd (binary coded decimal) numbers. This means that for these unsupported data types, you cannot express in the standard format your semantics (interpretation) of these values at the bit level. However omitting the data type interpretation, still introduces the data type name. Hence you can use this name to overload the operators, and do your intended synthesis. However by just porting the DFG format, you cannot explain others the intended semantics. Although the type definitions have no direct support for record–like data structures, the bit operations do allow you to extract or insert bitfields.

4.3 Numeric value versus bit pattern

Numeric values and bit patterns are semantically considered as two different domains, which are bridged by data type definitions. Without data type definitions the DFG can operate in the domain of numerical values. Constant values can be specified in the form of decimal numbers. Numerical operations as +, –, ×, ++, and ≤ or ≡ operate as mathematically expected. The behavior in this 'numerical value' domain corresponds to specifications like computer programming languages. Even 'low level' languages as C, do not define the behavior at the bit–level, but leave this (on purpose) machine dependent.

In the domain of bit patterns (bit vectors) the DFG can correctly pass around bit vectors as data values (tokens). Constants are specified in the form of strings of hexadecimal or octal digits. Bit operations as &, |, ~, bit–merge are valid in this domain.

Without the annotation with data types and data widths, numerical operations have undefined semantics on bit patterns (you cannot associate a value with them), and bit operations have undefined semantics on numerical values (you don't know the involved bit pattern). The addition of data types and widths can be seen as the first hardware design decisions. After adding them, numeric values are automatically converted to bit patterns and vice versa, hence numeric and bitwise operations can be freely intermixed. See Figure 8.

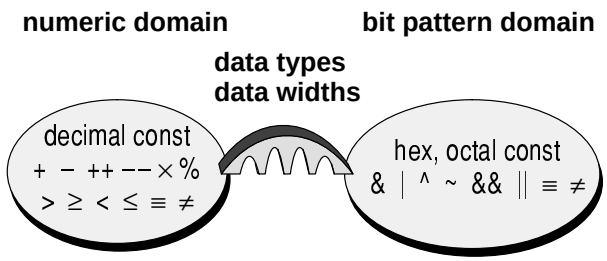

Figure 8. Numeric versus bit pattern domain

5. Textual Format

For an easy interface to various programming languages, a text (ASCII) based format is used. This has the additional advantage of easy transfer between different machines. The Lisp and EDIF syntax style using a pair of braces for each keyword ensures simple parsing: any LL–1 parser is strong enough, such as a recursive descent parser scheme. It furthermore allows for local and future extensions to the format, without disturbing already existing software (**both upwards and downwards compatibility**), and does not require a set of reserved words, forbidden as identifier.

The basic format is very simple: every statement forms a list. Any list starts with an opening brace and a keyword on which the application determines its interest in the list. The items of the list are names, numbers and other lists, and the list is terminated with a closing brace. If an application is not interested in the information attached to the keyword –or doesn't recognize the keyword– it can skip this list, without knowing anything about its (structured) contents, by just counting braces. Hence every tool or site is free to add more data for its own purpose. This property was considered highly important.

An unrestricted style of identifiers (all printable characters, unrestricted length, no reserved words) is chosen to simplify the translation of other languages into this format.

6. Timing

The format allows an accurate control over timing properties, such that scheduling algorithms can generate an efficient design, allowing options like chaining and multi–cycling. This timing information originates conceptually from two different sources:

The behavioural design specification will normally include timing restrictions such as minimum and maximum allowable delays between (mainly input and output) operations, and ordering relations between these. This information is represented in the format on additional timing edges. Furthermore data on desired clock speeds can be represented.

The module library will provide information regarding the operation nodes in the graph: real time delay of asynchronous modules, and number of clock cycles for synchronous modules together with leading and trailing real time delay and allowed clock speeds. Furthermore ripple delays can be specified, allowing a reasonable estimation of total delay when stacking operation nodes like adders and array multipliers.

7. Parameterization

The two main reasons for parameterization of the DFG are:

- Actual parameter assignment at the nodes of a graph is necessary to direct or control the generation of hardware operators.
- It should be possible to define a graph which is parameterized for (at least) the width of its edges.

Parameter passing concepts can lead to an enormous amount of extra syntax and constructs in a language. We felt that the DFG as exchange format, in contrast to a user interface language, would be better served with a syntactically minimal but still general enough scheme. Due to the keyword driven nature of the DFG format, extension remains possible of course. These considerations based the following choices:

- Nodes can have attached actual parameter assignments.
- Graphs can have formal parameter declarations. Together with the parameter assignment at nodes, this allows passing of parameters through a hierarchy of DFGs.
- All places where a number or identifier is allowed, also allow an expression.
- Parameters do not have a data type nor a structure. This greatly simplifies implementation. Whatever is assigned to a parameter, is just inserted when the parameter is used.
- Parameters are local to the graph where they are declared. Their names are unique in each graph. Therefor no scoping rules supporting concepts as name hiding need to be implemented.
- Parameters cannot influence the connection structure of a DFG. However they can change edge widths and edge data types.

8. Support

The DFG exchange format is described in a document, accurately defining the semantics, syntax, and context restrictions. Furthermore several tools are available to support development of software interfaces to the format: A skeleton parser and writer in two versions for the C and C++ implementation languages, a graph verification program which performs numerous checks on the correctness of the graph structure and type usage, and a graph drawing program with output on both X window screens and postscript printers.

9. Conclusion

A data flow graph exchange standard is described, which is used by several research organizations throughout Europe. It has a simple and intuitive token passing semantics, suitable for both architectural synthesis and formal verification.

Due to the braces oriented syntax, site dependent or application dependent extensions are allowed without disturbing other interfaces to the format, who do not know about these. Because both architectural synthesis and formal verification are still in the research phase this is an important property, allowing future extension. The syntax furthermore ensures simple parsing, and does not create a set of forbidden words for identifiers.

The true global data flow analysis and merge of data and control flow, lead to a maximal parallel design representation. As result, unsurpassed freedom for timing and allocation optimizations is obtained. The adopted data typing method is general, extendible and concise.

Current activities concentrate on the inclusion of (partial) synthesis results in the format, and a C++ programming interface and data structure to be used by all synthesis tools.

10. References

[Brayton88] BRAYTON, R.K., R. CAMPOSANO, G. DE MICHELI, R.H.J.M. OTTEN, AND J.T.J. VAN EIJNDHOVEN, "The Yorktown Silicon Compiler System," in *Silicon Compilation* , ed. D.D. Gajski, pp. 204–310, Addison–Wesley, 1988.

[Campos89] CAMPOSANO, R. AND W. ROSENSTIEL, "Synthesizing Circuits from Behavioral Specifications," *IEEE Trans on Computer–Aided Design*, vol. 8, no. 2, pp. 171–180, February 1989.

[Hilfin84] HILFINGER, P.N., *SILAGE: A Language for Signal Processing*, University of California, Berkeley, 1984.

[Jong91] JONG, G.G. DE, "Data flow graphs: system specification with the most unrestricted semantics", in *proc. of the European Conf. on Design Automation (EDAC)*, Amsterdam, The Netherlands, pp. 401–405, Feb. 1991.

[Knapp85] KNAPP, D.W. AND A.C. PARKER, "A unified Representation for Design Information," in *Proc. of the 7th International Symposium on Computer Hardware Description Languages and their Applications*, Tokyo, August 1985.

[Krol92] KROL, TH., AND J. VAN MEERBERGEN, C. NIESSEN, W. SMITS, J. HUISKEN "The Sprite Input Language: An Intermediate Format for High Level Synthesis" in *proc. of the European Conf. on Design Automation (EDAC)*, Brussels, Belgium, March 1992.

[Lis88] LIS, J.S. AND D.D. GAJSKI, "Synthesis from VHDL," in *Proc. of the IEEE International Conference on Computer Design 1988* , pp. 378–381, 1988.

[Marwed85] MARWEDEL, P., "The MIMOLA Design System: A design System which spans several Levels," in *Methodologies of Computer System Design* , ed. B.D. Shriver, pp. 223–237, North Holland, 1985.

[Pangrle87] PANGRLE, B.M., "A Behavioural Compiler for Intelligent Silicon Compilation," Thesis, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1987.

[Paulin91] PAULIN, P.G. AND A. JERRAYA, "SIF: an Interchange Format for the Design and Synthesis of High–Level Controllers", in *Digest of the 5th High–Level synthesis Workshop*, Bühlerhöhe, Germany, march 1991.

[Stok91] STOK, L. *Architectural Synthesis and Optimization of Digital Systems*, Thesis Eindhoven Univ. of Tech, Fac. Electrical Engineering, Eindhoven, The Netherlands, pp 160, 1991

[Veen85] VEEN, A.H., *The Misconstrued Semicolon: reconciling imperative languages and dataflow machines*. Thesis Eindhoven Univ. of Tech, Eindhoven, The Netherlands, 1985

[Walker87] WALKER, R.A. AND D.E. THOMAS, "Design Representation and Transformation in the System Architect's Workbench", in: *Digest of Technical Papers of the Int. Conf. on Computer Aided Design 1987*, pp. 166–169, 1987.