An Algebraic approach to Block Diagram Constructions

Yann Orlarey, Dominique Fober, Stéphane Letz
Grame, Centre National de Création Musicale
9 rue du Garet, BP 1185
69202 Lyon Cedex, France

March 15, 2002

Abstract

We propose an algebraic approach to block diagram construction as an alternative to the classical graph approach inspired by dataflow models. The proposed algebra is based on three binary operations of construction : sequential, parallel and recursive constructions. These operations can be seen as high level connection schemes that set several connections at once in order to combine two block diagrams to form a new one. Algebraic representations have interesting applications for visual languages based on block diagrams and are useful to specify the formal semantic of this languages.

1 Introduction

The dataflow approach proposes several well known models of distributed computations (see [2] and[1] for historical papers, and [6] and [3] for surveys) and many block diagram languages are more or less directly inspired by these models. Due to their generality, dataflow have been used as a principal for computer architecture, as model of concurrency or as high level design models for hardware [4], the semantic of these various models can be quite complex and depends of many technical choices like, synchronous or asynchronous computations, deterministic or non-deterministic behavior, bounded or unbounded communication FIFOs, firing rules, etc.

Because of these complexities, the task of defining the formal semantic of block diagram languages based on dataflow models is not trivial and the vast majority of our dataflow inspired music languages don't have an explicit formal semantic. Providing a formal semantic is not just an academic question. Because of the great stability of the mathematical language, it is probably the best chance we have to preserve the *meaning* of our tools over a long period of time, and therefore the musics based on them, in a world of rapidly evolving technologies.

To solve the problem we propose a block diagram algebra (BDA), an algebraic approach to block diagram construction as an alternative to the classical graph approach inspired by dataflow models. The idea is to use high level construction operations to combine and connect together whole block diagrams, instead of individual connections between blocks. Having defined a set of construction operations general enough to build any block diagram, the formal semantic can be specified in a modular way by rules, associated to each construction operation, that relate the meaning of the resulting block diagram to the meaning of the block diagrams used in the construction.

There are several techniques to describe the semantic of a program. Since we are mostly interested by *what* is computed by a block diagram and not so much by *how* it is computed, we will adopt a *denotational* approach, which describes the meaning of a program by the mathematical object it denotes, typically a mathematical function. Moreover, in order to make things concrete and to simplify the presentation, we will restrict ourself to the domain of real time sound synthesis and processing.

2 Representation of block diagrams

In the classical approach inspired by dataflow models, block diagrams are viewed as graphs defined by a set of blocks and a set of connections between these blocks.

In the algebraic approach block diagrams are viewed as terms of a formal language.

2.1 Graph representation of block diagrams

A block diagram can be represented as a graph $G = (N, C)$ where N is a set of node, i.e. the blocks of the diagram, and C the set of connections between these blocks.

2.1.1 Nodes

To each node $n \in N$ is associated a set of *input ports* $\mathbf{ip}(n)$ and a set of *output ports* $\mathbf{op}(n)$. A node with exactly one output port and no input port is called an *input*. A node with exactly one input port and no output port is called an *output*.

2.1.2 Connections

A connection $c \in C$ is a triplet $(n_1, n_2, (p_1, p_2))$ where $n_1 \in N$ and $n_2 \in N$ are respectively the source and destination node of the connection, and (p_1, p_2) the output port $p_1 \in \mathbf{op}(n_1)$ and input port used $p_2 \in \mathbf{ip}(n_2)$ of the connection.

Because we are essentially interested in the topological aspects of the graph, we suppose the semantic of the primitive blocks, including time based operations like delays, to be defined elsewhere. However, in order to simplify the transformation of cycles, it is useful to consider that a connection c have a delay property $\mathbf{dl}(c)$ such that $\mathbf{dl}(c) = 0$ when c transmits signals instantaneously, and $\mathbf{dl}(c) = 1$ when it transmits signals with a 1 sample delay.

2.1.3 Reasonable block diagram

A graph represents a *reasonable* block diagram, in term of real time signal processing, if every cycle of the graph contains at least one connection c such that $\mathbf{dl}(c) = 1$.

2.2 Algebraic representation of block diagrams

In the algebraic approach adopted here, block diagrams are viewed as terms of a language $\mathbb{D}$ described by the following syntactic rule :

$$\begin{array}{rcl} d \in \mathbb{D} & ::= & b \in \mathbb{B} \\ & | & \_ \\ & | & ! \\ & | & (d_1 : d_2) \\ & | & (d_1 , d_2) \\ & | & (d_1 \sim d_2) \end{array}$$

We suppose elsewhere defined a set $\mathbb{B}$ of primitive blocks corresponding to the basic functionalities of the system, and such as for each $b \in \mathbb{B}$ we know the number of input ports $\mathbf{ins}(b)$ and output ports $\mathbf{outs}(b)$. Among these primitive blocks we consider two particular elements called *identity* "_" and *cut* "!".

Here is an informal description of these elements as well as the three binary operations of composition we propose.

2.2.1 Identity "_" and Cut "!"

As shown by figure 1 *identity* "_" is essentially a simple wire and *cut* "!" is used to terminate a connection.

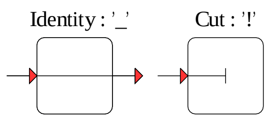

Figure 1: the _ and ! primitive

2.2.2 Sequential composition ":"

The sequential composition of B and C is obtained by connecting the outputs of B to the inputs of C according to the scheme of figure 2.

In its strict version, sequential connection is only allowed if the number of inputs of C is an exact multiple of the number of outputs of B : $\mathbf{outs}(B) * k = \mathbf{ins}(C)$ where $k \in \mathbb{N}^*$.

If $k = 1$ we can simplify the diagram as in figure 3.

It is convenient, but not essential in terms of generality of the algebra, to extend the sequential composition to the reverse case where the number of outputs of B is an exact multiple of the number of inputs of C : $\mathbf{outs}(B) = k * \mathbf{ins}(C)$. The inputs of C act as output bus for the outputs of B as in figure 4.

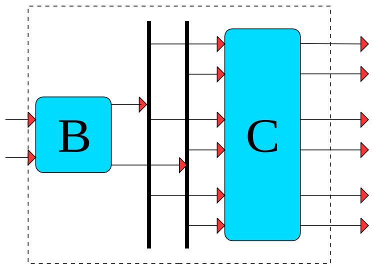

Figure 2: `(B:C)` sequential composition of B and C

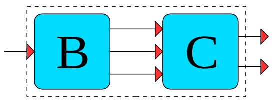

Figure 3: sequential composition of B and C when $k = 1$

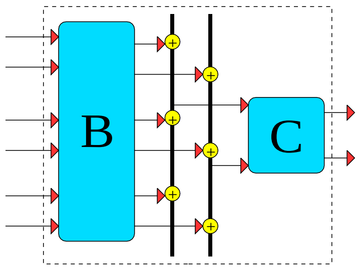

Figure 4: sequential composition when $\textbf{outs}(B) = k * \textbf{ins}(C)$

Another possible extension, but that we are not considering here, when the numbers of outputs and inputs are not related by an integer factor is described by figure 5.

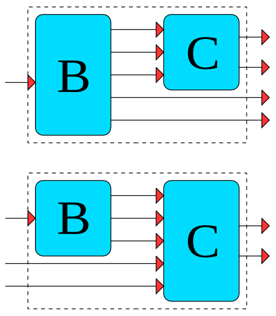

Figure 5: A second extension to sequential composition

2.2.3 Parallel composition “,”

The parallel composition of B and C is notated (B,C). It is represented figure 6.

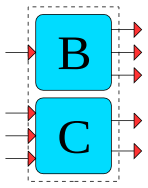

Figure 6: `(B,C)` parallel composition of B and C

2.2.4 Recursive composition “$\sim$”

Recursive composition, notated $B \sim C$, is essential for building block diagrams with feedbacks capable of computing signals defined by recursive equations. As shown by figure 7, the outputs of B are connected back to the inputs of C and the outputs of C are connected to the inputs of B. The operation is only allowed if $\textbf{outs}(B) \geq \textbf{ins}(C)$ and $\textbf{ins}(B) \geq \textbf{outs}(C)$. For practical reasons we incorporate directly into the semantic of the $\sim$ operation the 1-sample delays (represented by small yellow boxes on the diagrams) needed for the recursive equations to have a solution.

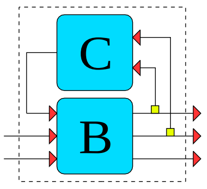

Figure 7: `(B~C)` recursive composition of *B* and *C*

2.3 Examples

We present two short examples of block diagram description. To simplify the notation and avoid too much parenthesis we will use the following precedence and associativity rules :

Precedence	*Associativity*	*Operator*	
3	left	$\sim$	rec
2	right	,	par
1	right	:	seq

2.3.1 Example 1

The example of figure 8 is typical of situation where you have an input stage, several parallel transformations combined together and an output stage. It is described by the following expression :

$$A : B, C, D : E$$

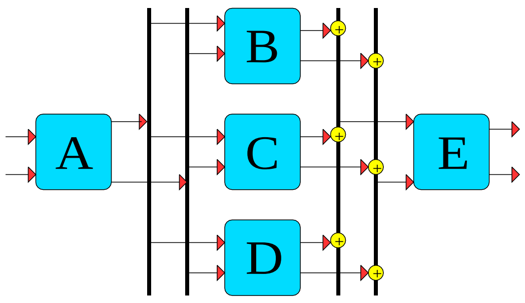

Figure 8: Several transformations in parallel

2.3.2 Example 2

The diagram of figure 9 is a little bit more complex to describe.

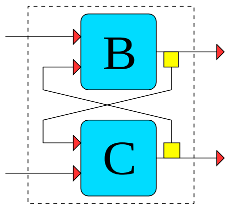

Figure 9: a typical block diagram with feedbacks

The first step is to rearrange the connections as in figure 10.

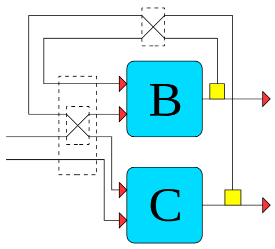

Figure 10: Same example after rearranging the connections

We see clearly now two places in the diagram where our wires have to cross. So the next thing to do is to describe an "X" block diagram allowing two wires to cross. The definition is given by the following formula and correspond to figure 11:

$$X = \_,\_ :!,\_,\_,!$$

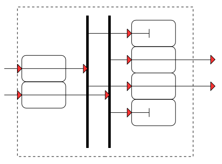

Figure 11: The block diagram $X = \_,\_ :!,\_,\_,!$ allows two wires to cross

The diagram is made of two selectors in parallel. The first selector : !,_ selects the second of its two inputs and the second selector : _,! selects the first of its inputs. This technique is easy to generalize to define any

$n \times m$ matrix of connections by composing in parallel m selectors, each selector being a parallel composition of one _ and $n-1$!.

Using X, the definition of the diagram of figure 10 is now straight forward :

$$(\_,X,\_ : B,C) \sim X$$

3 Well typed terms

As we have seen section 2, depending of the number of input and output ports of the blocks diagrams involved, not every operation is allowed. We can formalize these constraints as a small type system.

We define the type of a block diagram d to be defined by its number of inputs n and outputs m. We will write $d : n \rightarrow m$ to specify that diagram d has type $n \rightarrow m$. The type system is defined by the following inference rules :

$$(prim)\frac{}{b : n \rightarrow m}$$

$$(id)\frac{}{\_ : 1 \rightarrow 1}$$

$$(cut)\frac{}{! : 1 \rightarrow 0}$$

$$(seq)\frac{B : n \rightarrow m \quad C : m*k \rightarrow p \quad k \geq 1}{(B : C) : n \rightarrow p}$$

$$(seq')\frac{B : n \rightarrow m*k \quad C : m \rightarrow p \quad k \geq 1}{(B : C) : n \rightarrow p}$$

$$(par)\frac{B : n \rightarrow m \quad C : o \rightarrow p}{(B,C) : n+o \rightarrow m+p}$$

$$(rec)\frac{B : v+n \rightarrow u+m \quad C : u \rightarrow v}{(B \sim C) : n \rightarrow u+m}$$

For the rest of the paper we will assume well typed terms.

4 Number of inputs and outputs of a block diagram

We can now define precisely the **outs**() and **ins**() functions on well typed terms. For **outs**() we have :

$$\begin{array}{rcl}
\mathbf{outs}(\_) & = & 1 \\
\mathbf{outs}(!) & = & 0 \\
\mathbf{outs}(B : C) & = & \mathbf{outs}(C) \\
\mathbf{outs}(B,C) & = & \mathbf{outs}(B) + \mathbf{outs}(C) \\
\mathbf{outs}(B \sim C) & = & \mathbf{outs}(B)
\end{array}$$

And for **ins**() :

$$\begin{array}{rcl}
\mathbf{ins}(\_) & = & 1 \\
\mathbf{ins}(!) & = & 1 \\
\mathbf{ins}(B : C) & = & \mathbf{ins}(B) \\
\mathbf{ins}(B,C) & = & \mathbf{ins}(B) + \mathbf{ins}(C) \\
\mathbf{ins}(B \sim C) & = & \mathbf{ins}(B) - \mathbf{outs}(C)
\end{array}$$

5 Semantic of block diagrams

In this section we will see how to compute the semantic of a block diagram from the semantic of its components. We will adopt a *denotational* approach and describe this semantic by a mathematical function that maps input signals to output signals.

5.1 Definitions and notations

5.1.1 Signals

A signal s is modeled as discrete function of time

$$s : \mathbb{N} \rightarrow \mathbb{R}$$

For a signal s, we will write $s(t)$ the value of s at time t.

We call $\mathbb{S}$ the set of all *signals*

$$\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$$

5.1.2 Delayed signals

We will write x^{-1} the signal x delayed by one sample and such that :

$$\begin{array}{rcl}
x^{-1}(0) & = & 0 \\
x^{-1}(t+1) & = & x(t)
\end{array}$$

5.1.3 Tuple of signals

We will write :

1. $(x_1, \ldots, x_n)$: a n-tuple of signals of $\mathbb{S}^n$,
2. $()$: the empty tuple, single element of $\mathbb{S}^0$,
3. $(x_1, \ldots, x_n)^k$: the tuple $(x_1, \ldots, x_n)$ repeated k times.

5.1.4 Signal Processors

We define a signal processor p as a function from a n-tuple of signals to a m-tuple of signals :

$$p : \mathbb{S}^n \rightarrow \mathbb{S}^m$$

We call $\mathbb{P}$ the set of all signal processors :

$$\mathbb{P} = \bigcup_{n,m \in \mathbb{N}} \mathbb{S}^n \rightarrow \mathbb{S}^m$$

5.1.5 Semantic function

The semantic function $[\![.]\!] : \mathbb{D} \rightarrow \mathbb{P}$ associates to each well typed block diagram $d \in \mathbb{D}$ the signal processor $p \in \mathbb{P}$ it denotes. It is such that

$$[\![d]\!] = p : \mathbb{S}^{\mathbf{ins}(d)} \rightarrow \mathbb{S}^{\mathbf{outs}(d)}$$

5.2 The semantic function $[\![.]\!]$

The semantic function $[\![.]\!]$ is defined by the following rules

5.2.1 Identity

$$[\![\_]\!]\, x = x$$

5.2.2 Cut

$$[\![!]\!]\, x = ()$$

5.2.3 Sequential composition

case $\mathbf{outs}(B) * k = \mathbf{ins}(C)$

$$\begin{array}{rcl} [\![B:C]\!]\,(x_1,\ldots,x_n) & = & (y_1,\ldots,y_p) \\ \mathbf{where}\,(y_1,\ldots,y_p) & = & [\![C]\!](s_1,\ldots,s_m)^k \\ (s_1,\ldots,s_m) & = & [\![B]\!]\,(x_1,\ldots,x_n) \end{array}$$

case $\mathbf{outs}(B) = \mathbf{k} * \mathbf{ins}(C)$

$$\begin{array}{rcl} [\![B:C]\!](x_1,\ldots x_n) & = & (y_1,\ldots y_m) \\ \mathbf{where}\,(y_1,\ldots,y_p) & = & [\![C]\!](\sum_{j=0}^{k-1}(s_{1+j.m}),\ldots \sum_{j=0}^{k-1}(s_{m+j.m})) \\ (s_1,\ldots,s_m) & = & [\![B]\!]\,(x_1,\ldots,x_n) \end{array}$$

5.2.4 Parallel composition

$$\begin{array}{rcl} [\![B,C]\!]\,(x_1,\ldots,x_n,s_1,\ldots,s_o) & = & (y_1,\ldots,y_m,t_1,\ldots,t_p) \\ \mathbf{where}\,(y_1,\ldots,y_m) & = & [\![B]\!]\,(x_1,\ldots,x_n) \\ (t_1,\ldots,t_p) & = & [\![C]\!]\,(s_1,\ldots,s_o) \end{array}$$

5.2.5 Recursive composition

$$\begin{array}{rcl} [\![B \sim C]\!]\,(x_1,\ldots,x_n) & = & (y_1,\ldots,y_m) \\ \mathbf{where}\,(y_1,\ldots,y_m) & = & [\![B]\!](r_1,\ldots,r_v,x_1,\ldots,x_n) \\ (r_1,\ldots,r_v) & = & [\![C]\!](y_1^{-1},\ldots,y_{u\leq m}^{-1}) \end{array}$$

6 Generality of the BDA

The Block Diagram Algebra can be used to represent any *reasonable* block diagram. In the next paragraphs we present informally a general method to transform a graph representation of a block diagram into its algebraic representation.

In the last paragraph we will show that the BDA as the same expressive power of the Algebra of Flownomial.

6.1 Transformation of the graph representation

Graphs representing *reasonable* block diagrams can be transformed into algebraic terms by applying the following steps.

6.1.1 Marking of delayed connections

The first step of the transformation is to mark every connection with a delay like in the graph of figure 12. Connections with the same origin like the output of D receive the same mark (for example R1)

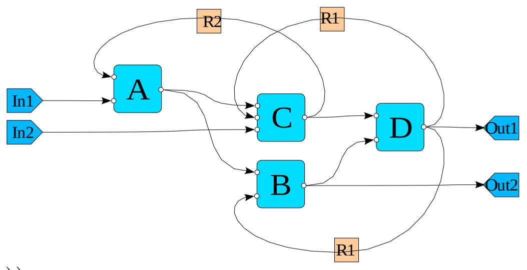

Figure 12: marking the connections with a delay

6.1.2 Opening of marked connections

The second step is to *open* every marked connection as in figure 13. Two new nodes are created for every marks

: a *recursive* output and a *recursive* input. The graph is now acyclic.

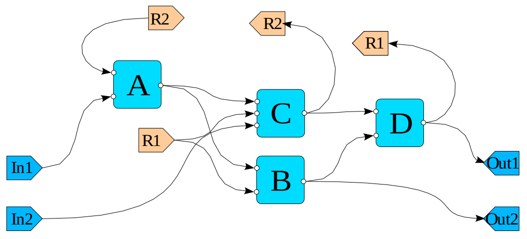

Figure 13: opening the marked connections

6.1.3 Topological sort

The third step is to make a topological sort in order to have on the first left-most column all the nodes that don't have a predecessor, then on the second column all the nodes that only have predecessors on the first column, etc. until the last column.

6.1.4 Rearranging inputs and outputs

Then we have to rearrange the inputs from top to bottom : $R_n,\dots,R_1,In_1,\dots,In_m$ and the outputs : $R_n,\dots,R_1,Out_1,\dots,Out_p$ as in figure 14. The order of the R_i doesn't matter provided it is the same for the inputs and the outputs.

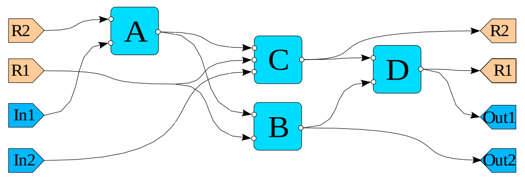

Figure 14: rearranging the inputs and outputs

6.1.5 Representing the acyclic graph

The next step is to code the acyclic graph. This is always possible because we can represent any $n \times m$ matrix of connection between two blocks using m selectors of the form $(!,\dots,!,\_,!,\dots!)$ in parallel.

Let call $X = \_,\_ :!,\_,\_,!$ and $Y = \_ : \_,\_$. X crosses two wires and Y split a wire in two. We can rearrange the connections of the graph of figure 14 as in figure 15 which corresponds to the term G :

$$G = \_,X,\_ : A,\_,\_ : C,B,! : Y,Y : \_,D,\_ : \_,Y,\_$$

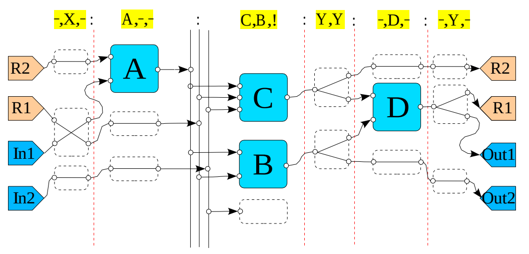

Figure 15: rearranging the connections

6.1.6 Final step

The final result corresponding to figure 16 is obtained by making a recursive composition using as many "_" as the number of recursive inputs-outputs (here 2) and then adding a final stage that remove these recursive connections for the outside :

$$G \sim (\_,\_) :!,!,\_,\_$$

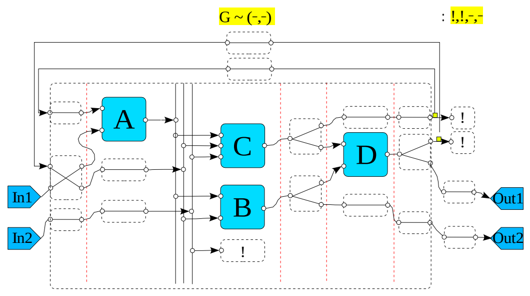

Figure 16: final step

6.2 Equivalence with the Algebra of Flownomials

We give here an indirect proof that the BDA can represent any block diagram by giving its equivalence with the Algebra of Flownomials (AoF). Proposed by

Gh. Stefanescu [5] the AoF can represent any directed flowgraphs (blocks diagrams in general including flowcharts) and their behaviors. It is based on three operations and various constants used to describe the branching structure of the flowgraphs. They all have a direct translation into our BDA as shown table 1.

AoF		BDA
par. comp.	$A \mathbin{+\!\!\!+} B$	A, B
seq. comp	$A.B$	$A : B$
feedback	$A \uparrow$	$(A \sim \_) : (!, \_^{\mathbf{outs}(A)-1})$
identity	I	$\_$
transposition	X	$(\_, \_) : (!, \_, \_, !)$
ramification	$\wedge_k^n$	$\_^n : \_^{n*k}$
	$\wedge_0$	$!$
identification	$\vee_n^k$	$\_^{n*k} : \_^n$

Table 1: Correspondences between the algebra of Flownomials and our block diagram algebra. Note : $\_^n$ is an abbreviation that means the composition of *n identity* in parallel.

7 Conclusion

The contribution of the paper is a general *block diagram algebra*, based on two constants and three operations, and its denotational semantic. This algebra is powerful enough to represent any block diagram while allowing a compact representation in many situations.

Algebraic representations of block diagrams have several interesting applications for visual programming languages. First of all they are useful to formally define the semantic of the language and, as stated in the introduction, there is a real need for such formalizations if we want our tools (and the musics based on them) to survive.

At a user interface level, algebraic block can be used in block diagram editor as an equivalent textual representation in addition to the graphic representation. They can be used also to simplify and enforce a structured representation of visual diagrams that is easier to follow and understand for the user.

Algebraic representations have also the advantage, compared to graph representations, to be easier to manipulate and analyze formally. They can be used as an adequate internal representation for compilers and optimizers that need to do smart things like abstract interpretation, specialization, partial evaluation, etc..

References

[1] J. B. Dennis and D. P. Misunas. A computer architecture for highly parallel signal processing. In *Proceedings of the ACM 1974 National Conference*, pages 402–409. ACM, November 1974.

[2] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland, 1974.

[3] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[4] Najjar, Lee, and Gao. Advances in the dataflow computational model. *PARCOMP: Parallel Computing*, 25, 1999.

[5] Gheorghe Stefanescu. The algebra of flownomials part 1: Binary flownomials; basic theory. Report, Technical University Munich, November 1994.

[6] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.