



An Algebraic approach to Block Diagram Constructions

Yann Orlarey, Dominique Fober, Stéphane Letz

► **To cite this version:**

Yann Orlarey, Dominique Fober, Stéphane Letz. An Algebraic approach to Block Diagram Constructions. Journées d'Informatique Musicale, 2002, Marseille, France. pp.151-158. hal-02158931

HAL Id: hal-02158931

<https://hal.archives-ouvertes.fr/hal-02158931>

Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 26, 2002

Abstract

We propose an algebraic approach to block diagram construction as an alternative to the classical graph approach inspired by dataflow models. This block diagram algebra is based on three binary operations : 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. Such algebraic representations have interesting applications for visual languages based on block diagrams. In particular they are very useful to specify the formal semantic of these 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). It have been used as a principal for computer architecture, as model of concurrency or as high level design models for hardware [4]. Many block diagram languages are more or less directly inspired by these models.

Due to their generality the semantic of these various dataflow models can be quite complex. This complexity depends on many technical choices like, synchronous or asynchronous computations, deterministic or non-deterministic behavior, bounded or unbounded communication FIFOs, firing rules, etc. For these reasons the task of defining the formal semantic of block diagram languages based on dataflow models is not trivial. This is probably why the vast majority of our dataflow inspired music languages have no 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 thus 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 whole block diagrams together, 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 in *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, 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.

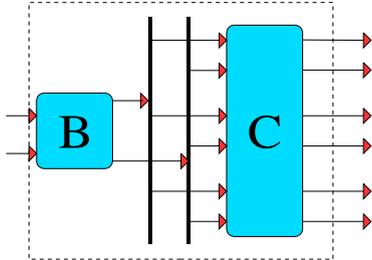


Figure 2: $(B:C)$ sequential composition of B and C

Another possible extension (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.

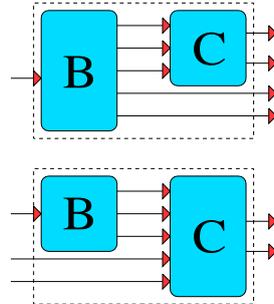


Figure 5: A second extension to sequential composition

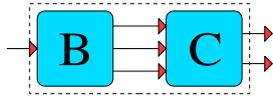


Figure 3: sequential composition of B and C when $k = 1$

2.2.3 Parallel composition “,”

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

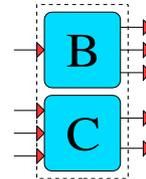


Figure 6: (B,C) parallel composition of B and C

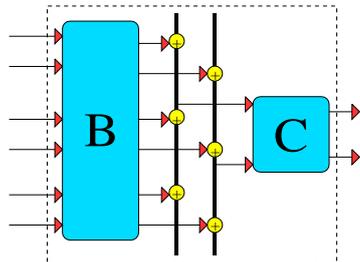


Figure 4: sequential composition when $\mathbf{outs}(B) = k * \mathbf{ins}(C)$

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 $\mathbf{outs}(B) \geq \mathbf{ins}(C)$ and $\mathbf{ins}(B) \geq \mathbf{outs}(C)$. For practical reasons we directly incorporate into the semantic of the \sim operation the 1-sample delays (represented by small yellow boxes on the diagrams) that are needed for the recursive equations to have a solution.

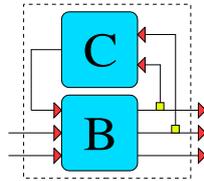


Figure 7: $(B \sim C)$ recursive composition of B and C

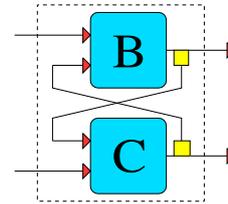


Figure 9: a typical block diagram with feedbacks

2.3 Examples

Here are two short examples of block diagram description. To simplify the notation and avoid too many 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 :

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

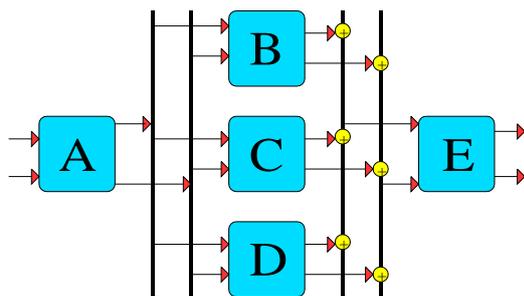


Figure 8: Several parallel transformations

2.3.2 Example 2

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

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

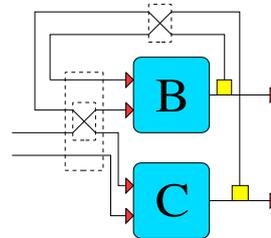


Figure 10: Same example after rearranging the connections

We clearly see 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 corresponds to figure 11:

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

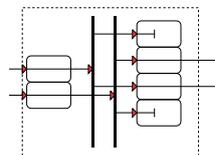


Figure 11: The block diagram $X = _ , _ : ! , _ , _ , !$ allows two wires to cross

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 in 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 determined 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 :

$$\begin{array}{c} (prim) \frac{}{b : n \rightarrow m} \\ (id) \frac{}{_ : 1 \rightarrow 1} \\ (cut) \frac{}{! : 1 \rightarrow 0} \\ (seq) \frac{B : n \rightarrow m \quad C : m * k \rightarrow p \quad k \geq 1}{(B : C) : n \rightarrow p} \\ (seq') \frac{B : n \rightarrow m * k \quad C : m \rightarrow p \quad k \geq 1}{(B : C) : n \rightarrow p} \\ (par) \frac{B : n \rightarrow m \quad C : o \rightarrow p}{(B, C) : n + o \rightarrow m + p} \\ (rec) \frac{B : v + n \rightarrow u + m \quad C : u \rightarrow v}{(B \sim C) : n \rightarrow u + m} \end{array}$$

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 precisely define the $\mathbf{outs}()$ and $\mathbf{ins}()$ functions on well typed terms. For $\mathbf{outs}()$ we have :

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

And for $\mathbf{ins}()$:

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

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 with 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 :

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

5.1.3 Tuple of signals

We will write :

1. (x_1, \dots, x_n) : a n -tuple of signals of \mathbb{S}^n ,
2. $()$: the empty tuple, single element of \mathbb{S}^0 ,
3. $(x_1, \dots, x_n)^k$: the tuple (x_1, \dots, 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 $\llbracket \cdot \rrbracket : \mathbb{D} \rightarrow \mathbb{P}$ associates to each well typed block diagram $d \in \mathbb{D}$ the signal processor $p \in \mathbb{P}$ it denotes. It is such that

$$\llbracket d \rrbracket = p : \mathbb{S}^{\text{ins}(d)} \rightarrow \mathbb{S}^{\text{outs}(d)}$$

5.2 The semantic function $\llbracket \cdot \rrbracket$

The semantic function $\llbracket \cdot \rrbracket$ is defined by the following rules

5.2.1 Identity

$$\llbracket _ \rrbracket x = x$$

5.2.2 Cut

$$\llbracket ! \rrbracket x = ()$$

5.2.3 Sequential composition

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

$$\begin{aligned} \llbracket B : C \rrbracket (x_1, \dots, x_n) &= (y_1, \dots, y_p) \\ \text{where } (y_1, \dots, y_p) &= \llbracket C \rrbracket (s_1, \dots, s_m)^k \\ (s_1, \dots, s_m) &= \llbracket B \rrbracket (x_1, \dots, x_n) \end{aligned}$$

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

$$\begin{aligned} \llbracket B : C \rrbracket (x_1, \dots, x_n) &= (y_1, \dots, y_m) \\ \text{where } (y_1, \dots, y_p) &= \llbracket C \rrbracket (\sum_{j=0}^{k-1} (s_{1+j.m}), \dots, \sum_{j=0}^{k-1} (s_{m+j.m})) \\ (s_1, \dots, s_m) &= \llbracket B \rrbracket (x_1, \dots, x_n) \end{aligned}$$

5.2.4 Parallel composition

$$\begin{aligned} \llbracket B, C \rrbracket (x_1, \dots, x_n, s_1, \dots, s_o) &= (y_1, \dots, y_m, t_1, \dots, t_p) \\ \text{where } (y_1, \dots, y_m) &= \llbracket B \rrbracket (x_1, \dots, x_n) \\ (t_1, \dots, t_p) &= \llbracket C \rrbracket (s_1, \dots, s_o) \end{aligned}$$

5.2.5 Recursive composition

$$\begin{aligned} \llbracket B \sim C \rrbracket (x_1, \dots, x_n) &= (y_1, \dots, y_m) \\ \text{where } (y_1, \dots, y_m) &= \llbracket B \rrbracket (r_1, \dots, r_v, x_1, \dots, x_n) \\ (r_1, \dots, r_v) &= \llbracket C \rrbracket (y_1^{-1}, \dots, y_m^{-1}) \end{aligned}$$

6 Generality of the BDA

The Block Diagram Algebra can be used to represent any *reasonable* block diagram. In the next paragraphs we informally present 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 has 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 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 in figure 12. Connections with the same origin like the output of D receive the same mark (for example R1)

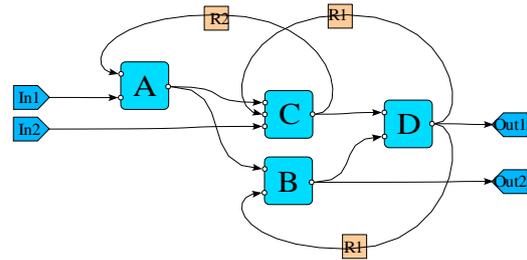


Figure 12: marking the connections with a delay

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 mark

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 in table 1.

AoF	BDA	
par. comp.	$A \# B$	A, B
seq. comp.	$A.B$	$A : B$
feedback	$A \uparrow$	$(A \sim _) : (!, _ \text{outs}(A)^{-1})$
identity	I	$_$
transposition	X	$(_, _) : (!, _, _, !)$
ramification	\wedge_k^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 also be used to simplify and enforce a structured representation of visual diagrams that is both easier to follow and understand for the user.

Algebraic representations also have 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.