# The CAPH Language, Ten Years After

Jocelyn Sérot, François Berry

# The CAPH language, ten years after. What did we do wrong ?

Jocelyn Sérot[1], Francois Berry[1]

Institut Pascal, UMR 6602 Université Clermont Auvergne/CNRS/SIGMA
{jocelyn.serot,francois.berry}@uca.fr

**Abstract.** This paper is a critical self-assessment of the CAPH dataflow-based programming language. We try to identify some design mistakes which could explain why the language and its associated toolset, despite some very innovative features, never received a wide acceptance.

## 1 Introduction

CAPH [1, 2] is a dataflow-based, domain-specific language for describing, simulating and implementing stream-processing applications. Applications are described as networks of purely dataflow actors exchanging tokens through unidirectional channels. From this description, the CAPH compiler can, as illustrated in Fig. 1 :

– simulate the behavior of the application,
– generate a software implementation in SystemC,
– generate an hardware implementation in VHDL, ready to be synthetized on a FPGA circuit.

The development of the CAPH language and associated toolset started in 2008. The initial motivations were very pragmatical[1]. Our research team had started to develop FPGA-based cameras for embedded vision applications and we immediately agreed on the fact that having to use RT-level languages, such as VHDL or Verilog, to program these devices would significantly hinder their usage, esp. by "software" programmers with no specific skills in hardware design. High-Level Synthesis was only emerging at this period and available tools were either too expensive or produced inefficient or platform-specific code. Our target applications, however, shared a particular feature : they had to operate "on the fly" on digital video streams coming directly from sensors and were limited to low level image processing. For this kind of application, the dataflow model of computation offers a very elegant solution to the synthesis problem, because it can be used both as a *programming* and an *implementation* model : actors can be described as finite state automata and channels as FIFOs, which can both be implemented directly and efficiently in hardware, without the need for a global control mechanism. The development of the CAPH language therefore

[1] For a detailed account on the origins of the CAPH project, see [4].

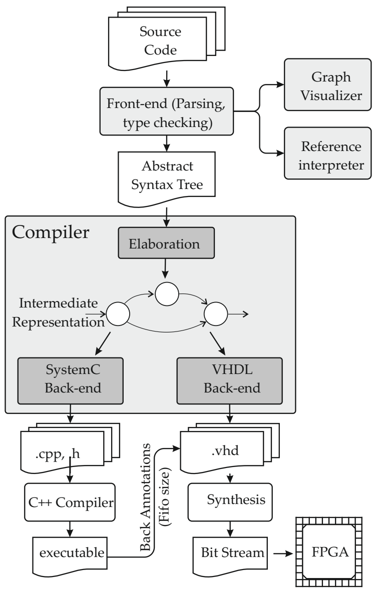


**Fig. 1.** The CAPH design flow

started with the very simple goal of allowing applications to be described as dataflow graphs in which the behavior of actors could be reduced to some kind of automaton. Compared to other similar projects[2], the two main distinctive characteristics were

- the possibility to systematically derive hardware implementations (by means of RTL transcription of actor behavior),
- the choice of a purely functional formalism for describing the structure of the dataflow networks.

Other features were added to the language latter with the general idea of increasing the *expressivity* of the language[3]. But, and contrary to what we expected, these additions did not help in increasing the audience of the language. In fact, a survey showed that most of users were only relying on the core, "historical", possibilities of the language, ignoring these extra features.

In this paper, we describe, in Sec. 3, two of these features and propose explanations for their non-adoption. In Sec. 4, we try to identify more general reasons why the language did not gain a wide acceptance, with the hope that the drawn conclusions can help language designers in the context of dataflow-based design. In order to be as self-contained as possible, the paper starts by a short presentation of the CAPH language in Sec. 2.

[2] The CAL language, for instance.
[3] Up to very recently. The latest version (2.9.0) was released in November 2018.

## 2 CAPH in a nutshell

The CAPH language is built from two layers : an actor description language (ADL) for describing the *behavior* of dataflow actors and a *network description language* (NDL) for describing the *structure* of dataflow networks. The ADL describes the behavior of individual actors as a set of *transition rules* involving pattern matching on input values and local variables. The NDL is a small, purely functional, polymorphic and higher-order language in which graphs are described by applying actors, interpreted as functions, to values representing wires.

A very simple CAPH program and its corresponding dataflow graph representation is given in Fig. 2. The application is a very simple $1 \times 3$ FIR filter[4]. The filter coefficients are defined as a global array constant line 1. The program uses two actors. The `d` actor, defined lines 3–6, is a *delay.* Given a stream of token $x_0, x_1, \ldots$, it produces the stream $0, x_0, x_1, \ldots$. The behavior of this actor is described using a internal variable `z` and a single activation rule, saying that whenever the actor reads a token carrying a value $x$ on its input `a`, it writes the value of the internal variable `z` on its output `c` and replaces this value by $x$. The `madd` actor, defined lines 8–12 describes a multiply-accumulate operation. Given a pair of tokens $x$ and $s$, it produces the token $s + x \times c$, where $c$ is given here as a (static) parameter. Lines 14–16 describes the global inputs and outputs of the program. In this version of the program, used for simulation, both input and output streams are read (resp. written) to files[5]. The input port `z` produces a stream of 0's. The dataflow graph (DFG) describing the filter itself is defined lines 18–22. The `d` actor is instanciated twice, producing respectively the one and two sample(s) delayed streams `x1` and `x2`. The `madd` actor is instanciated three times, implementing the classical "cascade" describing the filter. Each instance specifies a distinct value both for the parameter `c` and the actual IOs. The corresponding DFG is also given in Fig. 2.

## 3 Extra features

The program in Fig. 2 only uses the so-called "core" features of the language. In particular

– the involved actors are very simple and operate on basic types (signed, 8-bit integers),
– the application DFG is described in a low level, explicit manner, by simply naming intermediate wires.

In the sequel, we describe several reformulations of this program, each introducing a new feature of the CAPH language. In each case, we first try to

[4] This program is given only to illustrate some features of the language. It should not be viewed, in particular, as representative of the complexity of typical CAPH programs.

[5] A version intended for hardware synthesis will bind these inputs to process performing the physical IOs.

```
const coeff = [1,2,1] : signed<8> array[3];

actor d in (a:signed<8>) out (c:signed<8>)
  var z : signed<8> = 0
rules
| a:x -> (c:z, z:x);

actor madd (c:signed<8>)
   in (x:signed<8>, s:signed<8>)
 out (y:signed<8>)
rules
| (x:x, s:s) -> y:s+x*c ;

stream x:signed<8> from "sample.txt";
port z:signed<8> init 0;
stream y:signed<8> to "result.txt";

net x1 = d x;
net x2 = d x1;
net y1 = madd (coeff[0]) (x,z);
net y2 = madd (coeff[1]) (x1,y1);
net y = madd (coeff[2]) (x2,y2);
```

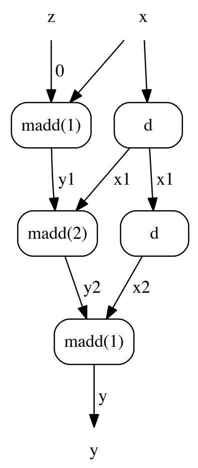


**Fig. 2.** A simple FIR filter described in CAPH

demonstrate the benefits of the feature and then speculate on the reasons why, despite this, it has not been adopted.

The reader must keep in mind that most of the explanations are very speculative because they were drawn from feedback from a very small set of users (less than a dozen) and from indirect observations[6].

## 3.1 Higher-order functions

A *higher-order* function (HOF) is a function accepting other functions as argument. A classical example is the `map` HOF, which takes a function $f$, a list of values $(x_1, \dots, x_n)$ and returns the list obtained by applying $f$ to each value $x_i$ :

$$\text{map } f \ (x_1, \dots, x_n) = (f \ x_1, \dots, f \ x_n)$$

In functional programming languages, HOFs play a key role by allowing the encapsulation of common, recurring patterns of computation. In the context of network description languages, HOFs naturally map to the concept of *higher-order wiring function* (HOWF). For example the program in Fig. 2 can be rewritten as in Listing. 1.1 (in which all unchanged parts have been denoted

[6] For example, if there's an obvious bug in the implementation of a feature which is not reported, we know that this feature has not been exercized...

as `...`). The `fir` HOWF is defined lines 2–7 as taking an array of coefficients `c`, a wiring function[7] `tap`, an input wire `x` and instanciated as specified line 9 to generates the "cascading" graph pattern shown in Fig. 2. In listing 1.1, the `tap` argument passed to the `fir` function is a single actor (`madd`). But – and this is where higher-orderness really shows its power – nothing prevents from passing a *function*. Listing 1.2, for example, gives another reformulation of the program in Fig. 2 in which the single actor `madd` is replaced by a *wiring function* `madd`, describing a subgraph composed of two distinct actors, `mult` and `add`. The corresponding DFG is given in Fig. 3.

**Listing 1.1.** A reformulation of the program in Fig. 2 using a higher-order wiring function

```
1  ...
2  net fir c tap x =
3    let x1 = d x in
4    let x2 = d x1 in
5    let y1 = tap (c[0]) (z,x) in
6    let y2 = tap (c[1]) (y1,x1) in
7    tap (c[2]) (y2,x2);
8
9  net o = fir coeff madd i;
```

**Listing 1.2.** Another reformulation of the program in Fig. 2 using the `fir` higher-order wiring function defined in Listing 1.1

```
...
actor mult (c: signed<8>)
  in (i:signed<8>) out (o:signed<8>)
rules
| i:x -> o:c*x;

actor add in (i1:signed<8>, i2:signed<8>)
          out (o:signed<8>)
rules
| (i1:x,i2:y) -> o:x+y;

net madd c (x,s) = add (s, mult c x);
...
net o = fir coeff madd i;
```

**Discussion** Despite the fact that they significantly increase the abstraction level and reusability of programs, HOWF do not seem to have been widely used by CAPH programmers, at least as a means of *defining* their own abstractions[8]. Here is a list of potential reasons :

[7] The reason for using this term is given below.
[8] The CAPH standard library provides several pre-defined HOWF.

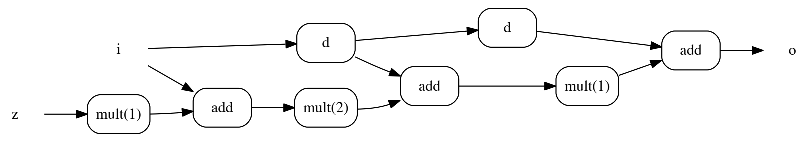


**Fig. 3.** The DFG corresponding to the program described in Listing 1.2

▷ The main usage of higher-order wiring functions is for encapsulating *graph patterns* and such patterns are only present in DFGs exhibiting some kind of *regularity*. Such regularity simply does not exist or is not worth to be encapsulated for "small" DFGs, containing only a few actors. This might be the case for the applications which have been developed with the tools.

▷ Large DFGs, describing applications at a finer grain level, essentially come from explicitely data-parallel formulations, for which specific solutions have already proposed to the "node explosion" problem. For example, in PREESM [5], replication is generally handled at the actor level. The LABVIEW dataflow-oriented IDE has a builtin factorisation mechanism to replicate actors.

▷ Higher-order functions ultimately requires a fully polymorphic type system, which can be disruptive for programmers not familiar with the subtleties of Hindley-Millner type inference and checking[9].

▷ The CAPH compiler systematically flattens the graph resulting from the instanciation of HOWFs. This is because all functions, including higher-order ones, are viewed as *specification*-level entities, not *implementation*-level ones. As a result, HOWF cannot be used to describe hierarchical graphs, in the sense used in the PiMM meta-model for example [6]. Having a direct interpretation of the former in terms of the latter would probably help for their adoption.

## 3.2 Algebraic data types

Algebraic data types (ADTs), also called *tagged unions*, allow values of different types to be mixed together by tagging them with a distinct label. Taking again our FIR example, consider the situation in which the data tokens could carry either real or complex values and that decision of how to process these values can only be made at runtime. A type for this kind of tokens could be defined with the following type declaration :

```
type sample =
    Real of signed<8>
  | Complex of signed<8> * signed<8>
```

[9] As a illustration, here's the type of the `fir` function given in Listing 1.2 : $\forall\alpha,\beta.\ \alpha\ \text{array}[\beta] \rightarrow (\alpha \rightarrow \text{signed}\langle 8\rangle \times \text{signed}\langle 8\rangle \rightarrow \text{signed}\langle 8\rangle) \rightarrow \text{signed}\langle 8\rangle \rightarrow \text{signed}\langle 8\rangle$.

This declaration says is that a value with type `sample` is

- either a real, encoded here as a 8-bit integer,
- or a complex, encoded here as a pair of 8-bit integers.

The associated tag (`Real` or `Complex`) is used to distinguish between these two cases. More generally speaking, the declaration of a variant type lists all possible “shapes” for values of that type. Each case is identified by a specific tag, called a *value constructor*, which serves both for constructing values of the variant type and inspecting them by pattern-matching. This is illustrated in Listing 1.3, which is a reformulation of the program in Fig.2 in which the type `signed<8>` has been replaced by the type `sample` defined above (again, all unchanged parts are denoted as `...`). The filter coefficients are here defined (line 1) as an array of values with type `sample`[10]. Both the `d` and `madd` actors now consume and produce values with type `sample`. The behavior of the `madd` actor is now described using four activation rules (lines 10–15). These rules respectively handle the situation in which both inputs are real, both inputs are complex and one input is real and the other complex. The actual computation is here supposed to be carried out by a function `madd_f`, taking the real and imaginary parts of the arguments and returning either a real or a complex value[11].

**Listing 1.3.** A reformulation of the program in Fig. 2 using algebraic data types

```
1  const coeff = [Complex(1,0), Complex(2,1), Complex (1,1)]
2    : sample array[3];
3
4  actor d in (a:sample) out (c:sample) ...;
5
6  actor madd (c:sample)
7     in (x:sample, s:sample)
8   out (y:sample)
9  rules
10 | (x:Real x, s:Real s) -> y:Real (madd_f(s,0,x,0,c))
11 | (x:Complex (xr,xi), s:Complex (sr,si))
12                      -> y:Complex(madd_f(s_r,s_i,x_r,x_i,c))
13 | (x:Complex (xr,xi), s:Real s)
14                      -> y:Complex(madd_f(s,0,x_r,x_i,c))
15 | (x:Real x, s:Complex (sr,si))
16                      -> y:Complex(madd_f(s_r,s_i,x,0,c);
17
18 stream x:sample from ”sample.txt”;
19 port z:sample init (Real 0);
20 stream y:sample to ”result.txt”;
21 ...
```

[10] With the convention, for simplicity, that the real coefficients are stored in the real parts.

[11] The code of this function has not been reproduced in Listing 1.3 for simplicity.

Algebraic data types can be polymorphic, *i.e.* they can be parameterized over (an)other type(s), called the *argument types(s)*. For example, we could have defined the type `sample` as follows

```
type $t sample =
    Real of $t
  | Complex of $t * $t
```

where `$t` can be any type suitable for encoding the real and imaginary parts, so that

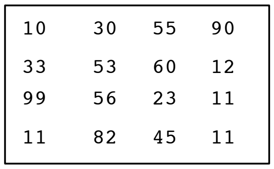
- the program of Listing 1.3 can be rewritten by replacing all instances of type `sample` by `signed<8> sample`,
- another version of the program, in which the real and imaginary parts are encoded, let say, as 32-bit float values, can be readily obtained by replacing, in the same program, the type `sample` by `float32 sample`.

Algebraic data types were originally introduced in CAPH to support the "data is control" concept. The idea that all the information required to interpret data streams must be embedded in the transported tokens. For example, images are encoded in CAPH using the following type :

```
type $t img = SoI | EoI | SoL | EoL | Pixel of $t
```

Tokens with values `SoI` and `EoI` (resp. `SoL` and `EoL`) are control tokens indicating the start and end of images (resp. lines) and pixels are carried by tokens having values `Pixel v`, so that, for example, the $4 \times 4$ image of Fig. 4 may be represented by the following stream of tokens:

```
SoI, SoL, Pixel(10), Pixel(30), Pixel(55), Pixel(90), EoL,
     SoL, Pixel(33), Pixel(53), Pixel(60), Pixel(12), EoL,
     SoL, Pixel(99), Pixel(56), Pixel(23), Pixel(11), EoL,
     SoL, Pixel(11), Pixel(82), Pixel(46), Pixel(11), EoL,  EoI
```



| | | | |
|---|---|---|---|
| 10 | 30 | 55 | 90 |
| 33 | 53 | 60 | 12 |
| 99 | 56 | 23 | 11 |
| 11 | 82 | 45 | 11 |

**Fig. 4.** A $4 \times 4$ image

With this representation, the dimensions of the images are explicitly contained in the token stream and hence no global control and/or synchronization is needed, which both allows the definition of size-generic actors and significantly eases the generation of RTL code. Moreover, it can be used to encode not only images but arbitrarily structured data.

**Discussion** Except for the `img` type described above[12], the use of ADTs seems to have been limited in programs. Again, there are several possible explanations.

▷ Historically, most of dataflow-based programming languages have only supported "flat", unstructured data, essentially viewing tokens as "black boxes", the interpretation of which was left to the actors themselves. As a result, programmers in the related fields are not familiar with the typing concepts and mechanisms used by functional programming languages – from which CAPH borrowed the notion of polymorphic ADT[13].

▷ The idea of encoding the size of the manipulated data structures *within* the token stream is in strong contrast with other incarnations of the dataflow MoC, such as PSDF [7] and PiSDF [6], in which these dimensions are explicitely and separately specified as *parameters* passed to the concerned actors during a specific configuration step. Our experience shows that for programmers used to model applications with the latter kind of MoC, the former idea is often disruptive.

▷ Moreover, this idea seems to imply that the CAPH MoC is a purely dynamic one and this may have taken away programmers requiring a more static MoC. Even if CAPH *allows* specification of DDF applications, it can also be used to describe application obeying to the SDF (Synchronous Dataflow) or CSDF (Cyclo Static Dataflow) MoCs.

▷ The initial choice of representing images using ADTs was maybe a bad idea since it actually forces the programmer to delve into the syntax and semantics of ADTs even for trivial image processing applications. Having a pre-defined, dedicated type for images with ad-hoc syntax elements to access the individual pixels might have been preferable[14].

## 4 Lessons learned

Several reasons can be invoked to explain why CAPH did not gain a wide acceptance as a programming language. We believe that these reasons can be related to three key questions that we probably have overlooked.

The first question concerns the **problems** the language is supposed to tackle.

The features discussed in Sec. 3.1 and 3.2[15] are undoubtedly powerful features confering to CAPH a distinctive position on the landscape of dataflow-based programming languages. They can also be viewed as a good example of the benefit

[12] Which is predefined in the standard library.

[13] A similar concept exists in C++-11, under the name "variant" but it was introduced recently and does not seem to be widely used.

[14] Moreover, representing images as lists of lists, as described above, has a very "lisp-ish" flavor which seems to induce strong repulsive reactions on certain kinds of programmers.

[15] And a few others not presented here, such as higher-order actors and dependent types for example.

of cross-fertilization between scientific domains (hardware design and programming language theory here). The problem is just that they were introduced not because there was a need for them expressed by the users, but because we, the designers of the language, thought there was.

Innovative and disruptive concepts may have an interest if the goal is to foster scientific or engineering understanding – which may perhaps later catalyze development of future languages – but, as long as the primary goal is to gain wide acceptance, these concepts should be introduced only if they solve a clearly identified problem or bottleneck in the existing design flow.

In this light, if we had the opportunity to restart the project from the beginning, we would probably devote more time to "low level" and pragmatic issues such as :

1. FIFO size minimisation whenever possible (the actor classification and static computation of these sizes for SDF DFGs only appear in version 2.9),
2. automatic retiming / pipelining for actors with long critical paths,
3. support for "soft" actors, written in C or C++ (and implemented as either as soft cores or on the HPS on the target FPGA),
4. support for common, cheap target platforms.

(1+2) would have helped breaking the "adoption barrier" among the hardware designers. Within a community which is much more concerned with resource usage and performance than the software one, a 50% overhead is often not acceptable, whatever the associated gain in productivity [8], as it is in sofware, by contrast.

(3) would have allowed to use CAPH as a high-level dataflow modeling tool within which the behavior of actors could be specified using a sequential imperative language such as C or C++. Of course, the scalability of the derived implementations remains low in this case because it is ultimately limited by the number of cores that can be instanciated on the target FPGA. But, with this approach, users can start by writing simple programs using their favorite programming language to specify actor behavior and then, if required, gradually switch to more hardware-friendy descriptions, such as those currently used in CAPH (a kind of Trojan horse, in a sense).

(4) Because such a support is, by essence, platform-specific, we did not provide one in the CAPH distribution (we had one, of course, but for our own needs, targeting a custom board designed in our laboratory). The idea was that CAPH was essentially platform-agnostic and that the so-called *board support packages* (BSPs) should be provided by the user. This makes sense of course when the target board is a specialized board (using for example specialized hardware drivers for image acquisition). But for the casual user, who simply wants to experiment with the language, the required effort is simply too high.

The second question concerns the **audience** of the language.

There clearly was an ambiguity, right from the start, concerning this audience. Was CAPH targeting in priority hardware of software designers ? In the first case, the language should have been presented as a way of increasing the productivity by, for instance, removing the hassle of explicit synchronisation between computation units and extending the data abstraction levels compared to classical HDLs such as VHDL or Verilog HDLs. In the second case, it should preferably have been presented as an introductory path to FPGA design, making HDL coding unnecessary whenever dataflow is an appropriate MoC. This question is of great practical importance because hardware and software designers, as far as our experience has shown us, do not seem to put focus on the same concerns. Many of the former view expressivity as a secondary concern and are reluctant to the adoption of any tool increasing productivity as long as the price to pay is a decrease in performances. Many of the latter are not ready to give up the "good old sequential imperative" way of thinking and are simply waiting for C++ HLS because they view efficiency and resource optimisation as a secondary concern.

The third and last question is not specific to CAPH and probably concerns all dataflow-based programming languages and frameworks. It has to do with the **invasive** nature of the proposed language and toolset.

In many cases, the dataflow model is only used to give a very coarse grain formulation of an application, in which the actors implement complex functions, even algorithms, and the edge essentially carry "opaque" packets of data (full images for example). This interpretation limits the shift in programming paradigm imposed to the programmer, who essentially continues to think in a imperative way[16]. Going to lower granularity requires that the semantics of the underlying MoC is exposed. Many (most) programmers are reluctant to this : unless they have no other choice, they frequently think that the price to pay is too high[17].

The situation is even worse if using a dataflow model requires a *complete* reformulation of the algorithm. Unfortunately, this is exactly what CAPH does.

In this light, we'd better have provided, right from the start, a way to use actors written in C or C++. This would have made possible to introduce the language in a less invasive and hence more attractive way, as follows :

1. describe how CAPH can produce a software implementation from a set of actors written in C or C++,
2. exhibit some typical, performance critical, applications for which a few actors are acting as bottlenecks,
3. show how these actors – which are likely to be associated to low level compute-intensive operations – can be rewritten using CAPH actor description language,
4. demonstrate that the resulting effort/gain ratio is significative (compared to direct re-coding in VHDL or Verilog).

On a technical side, for this approach to be effective, some work should have been devoted in developing a general mechanism / framework allowing a dataflow application, composed both of software actors, running on a soft core

[16] The DFG is in this case nothing more than a call-graph in disguise.

[17] To be fair, the subtleties in the taxonomy of dataflow-based MoCs, does not help.

for example, and hardware actors, implemented on a reconfigurable part of an FPGA, to seamlessly exchange tokens. This task has been in the "TODO" list of the CAPH project for years. Not giving it a high priority is likely to be one the worst mistake we made.

**Acknowledgements** Some ideas developed in this paper stem from numerous and fruitful discussions held during visits at the IETR laboratory in Rennes, in particular with M. Pelcat, K. Desnos and J.F. Nezan.

## References

1. The CAPH software and reference manual, http://caph.univ-bpclermont.fr
2. Sérot, J., Berry, F.: High-Level Dataflow Programming for Reconfigurable Computing. In: 2014 International Symposium on Computer Architecture and High Performance Computing Workshop, pp. 72–77, Paris (2014)
3. Sérot, J.: The CAPH Reference Manuel. http://caph.univ-bpclermont.fr/dist/caph-lrm.pdf
4. Sérot, J.: CAPH - A bit of history. http://caph.univ-bpclermont.fr/papers/misc/caph-history.pdf
5. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.-F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In 6th European Embedded Design in Conferenece on Education and Research, pp. 36–40, Milan (2014)
6. Desnos, K., Pelcat, M., Nezan, J.-F., Bhattacharyya, S.; Aridhi, S.: PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration. SAMOS XIII, Samos (2013)
7. Bhattacharya, B., Bhattacharyya S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing, 49(10), pp 2408–2421, 2001
8. Pelcat, M.: Models, Methods and Tools for Bridging the Design Productivity Gap of Embedded Signal Processing Systems. *Habilitation á Diriger des Recherches.* U. Clermont, 2016.