Hardware communication refinement in digital signal processing, modelling issues

Sylvain Huet, Emmanuel Casseau, Olivier Pasquier, Sébastien Le Nours

To cite this version:

Sylvain Huet, Emmanuel Casseau, Olivier Pasquier, Sébastien Le Nours. Hardware communication refinement in digital signal processing, modelling issues. 2008. hal-00332397

HAL Id: hal-00332397
https://hal.science/hal-00332397

Preprint submitted on 20 Oct 2008

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.

Hardware Communication Refinement in Digital Signal Processing, Modelling Issues

Sylvain Huet, Emmanuel Casseau
Université de Bretagne Sud
LESTER Lab CNRS FRE 2734
56321 Lorient Cedex, France
firstname.lastname@univ-ubs.fr

Olivier Pasquier, Sebastien Lenours
Polytech'Nantes
IREENA
BP50609, 44306 Nantes Cedex 3, France
firstname.lastname@polytech.univ-nantes.fr

Abstract

In this paper we present the different modelling problems which a Digital Signal Processing (DSP) application designer has to tackle while refining an abstract specification relying on coarse grain data (e.g. matrices) toward a hardware implementation model relying on fine grain data (e.g. scalar). To address this problematic, we propose a modelling framework which can be used to refine an algorithm specified with coarse grain interfaces to a form which allow, from the functionnality point of view, to model all its fine grain hardware implmentation.

1 Introduction

The classical approach for designing complex DSP applications is based on a top-down refinement flow where an initial abstract specification of the application is progressively and hierarchically decomposed into interacting subsystems.
There are many paths leading from the specification of a system to its implementation. To mark it out, some researchers have established graphical taxonomies which can help the designers to analyse their designs or to define the path which best suits their needs. The Y chart model due to Gajski [Gaj87] classifies the abstractions of a system on structural, behavioural and geometrical axes. Nevertheless, it lacks a representation of time and data abstractions. This is the reason why Ecker et al. introduce the Design Cube [EH92], where a design flow can be categorized according to the three following axes: timing, values, view. Seven years later, Jantsch et al. introduced the Rugby Model [JKH99]: they extend the notions of timing, values, view and add a fourth dimension, communication, to be able to classify modern hardware software design flows. The idea of the Design Cube has also been updated by Thabet et al. [TGCM04] with the aim at defining a communication refinement flow which includes data type refinements. This dimension is especially important in the context of hardware refinement. Actually, DSP application specifications often rely on abstract data types (e.g. matrix) whereas their hardware implementations work on scalar arithmetic operators in a parallel way. It results in a wide variety of implementation alternatives, each of them characterized by temporal performances (latency, throughput) and area costs (computation unit, memory unit).
To compare these implementations or to optimize the system during its refinement, it is interesting to have fine grain transactional models. For example, Filo et al. [FKCM93], Coussy et al. [CBM03] demonstrate in the High Level Synthesis (HLS) context that the adaptation of the fine grain communication patterns between the different components of a system can reduce the latency and the memory costs. In this particular context, it would be interesting to introduce the analysis of the fine grain communication patterns before the HLS process.

The paper is organized as follows. Section 2 presents the domain of application and the hardware implementations we address with our modelling framework. Section 3 presents the modelling framework. Section 4 gives some examples of application. At last we conclude in section 5.

2 Problem formulation

2.1 Application specification and definitions

In the context of this paper, we focus on DSP applications which can be specified as ***Synchronous Data Flow (SDF) programs*** [LM87]. These programs consist of directed graphs where each node represents an actor and each arc a signal path. The

number of data samples produced or consumed by each node on each invocation is known a priori.
An ***actor*** is specified by an ***algorithm*** and ***input and output interfaces***. In this paper, we consider that the interfaces are composed of communication ports each of them corresponding to a data token of the firing rules, i.e. as soon as all the input ports received one data token, the algorithm is fired and then all the output ports produce one data token. For example, a coarse grain matrix product function is composed of an input interface which has two ports to receive the operand matrices and of an output interface which has one input port where the result matrix is produced.
Each port of the interfaces receives a particular ***data type***. We classify the data types in two categories. Firstly the ***fine grain data*** which are operands or result of an operator used for the hardware implementation of an algorithm. In the context of this paper, a fine grain data is a scalar. Secondly the ***coarse grain data*** which are conceptual clusterings of fine grain data. Matrices, vectors, are common coarse grain data used to specify DSP applications.
Therefore an interface is either a ***fine grain interface*** if all its ports exchanged fine grain data or a ***coarse grain interface*** if at least one port exchanges a coarse grain data.
The SDF specifications introduce regular and iterative behaviours. These characteristics allow to define the algorithmic iteration concept. An ***algorithmic iteration*** is composed of the set of the input values and the set of resulting output values obtained by the algorithm firing. An algorithmic iteration is identified by an iteration number which is incremented by one each firing. Let us consider the algorithm f.
Let $IN^i = \{in_1^i, in_2^i, \ldots, in_m^i\}$ be the set of the values on the input interface of the algorithm f at the iteration i.
Let $OUT^i = \{out_1^i, out_2^i, \ldots, out_n^i\}$ be the set of the values on the output interface of the algorithm f at the iteration i, obtained by the relation $OUT^i = f(IN^i)$.
The algorithmic iteration number i of f is the couple (IN^i, OUT^i).
An algorithmic iteration is a logical organization of the data productions and consumptions.

2.2 Hardware implementation

According to the implementation of the algorithm, from the time line point of view, the execution of the algorithm can have different algorithmic iteration organisation schemes. Figure 1 presents the four usual ones. A box represents the time window in which the inputs or outputs are produced.

- **(a)** refers to the case where the outputs production time window of a given iteration coincide with its inputs consumption time window,
- **(c)** refers to the case where the output time window of a given iteration is shifted one iteration period to the corresponding input time window,
- **(b)** is the intermediate case between (a) and (c),
- **(d)** refers to the case where the output time window of a given iteration is shifted at least one iteration period to the corresponding input time window.

Cases (a) and (b) can present ***input output overlapping***, that is to say that given an iteration, some outputs can be produced before all the inputs arrived.

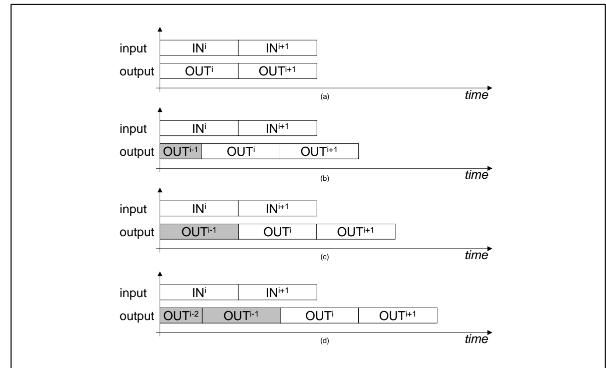

Figure 1: algorithmic iteration from timeline point of view

These four schemes are classical behaviours of hardware DSP algorithm implementations. Two difficulties appear to model these behaviours.

Functionality From the functional point of view, modelling the behaviours of patterns from (a) to (b) is non trivial. Indeed, the hardware implementation can lead to input output overlapping. For example, a matrix product algorithm implementation can produce fine grain outputs as soon as the corresponding left hand side operand row and right and side operand column are available. In such cases, it is not possible to use the original coarse grain algorithm specification to compute the outputs according to the inputs. Unlike the previous schemes, from the functional point of view, the others patterns are trivial to model, indeed the outputs of a given iteration are produced when all the corresponding inputs have already been received. Thus

it is possible to use the original coarse grain synchronized algorithm to compute the outputs.

Pipeline An implementation is pipelined if at a given instant, there is an overlapping between the inputs of one iteration and the outputs of another iteration. To model this effect, i.e. cases from (b) to (d), an iteration memory effect has to be introduced.

In the rest of this paper, cases (a), (b), (c) and (d) refer to the corresponding figures on figure 1.

The next section proposes a modelling framework which can be used to model cases from (a) to (d).

3 Modelling

To model all input output overlapping potential combinations of a given algorithm, i.e. all partial orders of the algorithm, we introduce an `iteration` object which has fine grain input and output interfaces. To model intermediate cases between coarse grain and fine grain interfaces, it is possible to add data grain wrappers to a fine grain interface, e.g. a vector to scalar wrapper for the inputs or scalar to vector wrapper for the outputs.

An `iteration_list` object is also introduced to manage the `iteration` objects, e.g. to manage the pipeline and to handle the inter-iterations dependencies. Before presenting these two objects, we present in the following sub-section 3.1 the algorithm model we use in these objects and in subsection 3.2 we present a method which can be used to extract the sets F_o and F_a introduced in subsection 3.1.

3.1 The algorithm model

In this paper we focus on DSP algorithms. These ones can have an inter-iterations memory effect, that is to say the algorithm can use past information to compute the current one. This is the consequence of the z^{-1} operators. We introduce the possibility to model this memory effect with fine grain variables which we call ageing variables. Consider the algorithm f.

Let $IN^i = \{in_1^i, in_2^i, \ldots, in_m^i\}$ be the set of the values of the fine grain inputs of the algorithm f at the iteration i.

Let $OUT^i = \{out_1^i, out_2^i, \ldots, out_n^i\}$ be the set of the values of the fine grain outputs of the algorithm f at the iteration i.

Let $A^i = \{a_1^i, a_2^i, \ldots, a_k^i\}$ be the set of the values of the ageing variables of the algorithm f at the iteration i.

From the functional point of view we have

$$(OUT^i, A^{i+1}) = f(IN^i, A^i)$$

However, this definition can not be applied in cases with input output overlapping. Indeed in such cases it is necessary to compute at least one subset of OUT^i in function of a subset of A^i and IN^i. To address such cases we define the two following algorithms sets:

Let $F_a = \{f_{a,1}, f_{a,2}, \ldots, f_{a,k}\}$ be the set of the algorithms which compute respectively $\{a_1^i, a_2^i, \ldots, a_k^i\} \forall i$.

Let $F_o = \{f_{o,1}, f_{o,2}, \ldots, f_{o,n}\}$ be the set of the algorithms which compute respectively $\{out_1^i, out_2^i, \ldots, out_n^i\} \forall i$.

The problems are:

(1) to find the smallest possible start spaces of these algorithms which constitute the smallest possible synchronisation grain,

(2) to find their expressions.

3.2 How to extract F_o and F_a?

To solve the first problem, the initial algorithm has to be analysed to determine the fine grain input output dependencies. To solve the second one it is possible to envisage a transformation of the initial algorithm specification or even to keep it untransformed for algorithms which have no inter iteration dependencies. In theses later cases the algorithm is duplicated for each element of F_o and F_a.

Nevertheless we use a method which formally solve theses two problems for algorithms which can be represented as a Fine Grain Signal Flow Graph (FGSFG).

A Signal Flow Graph (SFG) is a polar oriented acyclic graph $SFG(V, E)$ where:

- $V = \{v_0, \ldots, v_n\}$ is the set of the operation nodes, v_0 is the source node, v_n is the sink node. The operations can be arithmetic, data, logic or delay.

- $E = \{e_{ij}\}$ is the set of edges which represents the dependencies between the operation nodes v_i and v_j such that the operation v_j can start iff the operation v_i is completed.

The interfaces of the SFG are:

- the input nodes $v_i \in IN_{SFG} \subset V$ which represent data produced toward the SFG

- the output nodes $v_i \in O_{SFG} \subset V$ which represent data produced by the SFG.

The ageing variable nodes, $v_i \in A_{SFG} \subset V$ are the data operations nodes which have for unique predecessor a delay operation node.

A SFG and a Data Flow Graph (DFG) differ from the delay operation which is used in a SFG to model

inter algorithmic iteration dependencies.
A FGSFG is an SFG where all the operation nodes are fine grain operators.
To generate the algorithmic expressions and to extract the dependencies of each $v_i \in O_{FGSFG}$ and $v_i \in A_{FGSFG}$, we use an object oriented model of the FGSFG and do a recursive call of a polymorphic code generation method, i.e. which is specific to each kind of operation nodes, e.g. adder, multiplier, delay.
This method is called on each node $v_i \in O_{FGSFG}$ and each node $v_i \in A_{FGSFG}$ of the FGSFG. The recursion tree stops on delay operations, input operations, and constant data nodes. Figure 2 presents the code generation method of an adder operator.
We use GAUT's [gau] FGSFG generator, based on GCC [gcc], to extract a FGSFG from an algorithm.

```
String get_expression(NodeList nL) {

  String s ="";

  // for each predecessor of the current node
  foreach node n of pred(this)

    s += n.get_expression(nL);

    if n is not the last element of pred
      s+= "+"
    end if
  end foreach

  return( s );
}
```

Figure 2: adder get_expression algorithm

```
Class iteration< type >

Private Attributes
        matrix< bool > *        m_dep
        matrix< bool > *        v_in_pre
        matrix< type > *        v_in_val
        matrix< bool > *        v_out_pre
        matrix< type > *        v_out_val
        matrix< bool > *        v_out_con

Public Member Functions
        bool    refresh ()
        bool    in_exists (int ref)
        bool    out_exists (int ref)
        bool    put (type val, int ref)
        type    get (int ref)
        bool    is_consumed (int ref)
        bool    is_consumed ()
        bool    is_in_ageing (int ref)
        bool    is_out_ageing (int ref)
```

Figure 3: iteration object

3.3 The iteration object

The figure 3 presents the `iteration` object. This object manages the sets IN^i, OUT^i, A^i, A^{i+1} and applies the function f or the functions of F_a or F_o for the input output overlapping cases. The two following paragraphs present its attributes and methods.

3.3.1 Attributes

- `m_dep`

$$\texttt{m_dep}^i = \begin{array}{c|ccc:ccc|} & out_1^i & \dots & out_n^i & a_1^{i+1} & \dots & a_k^{i+1} \\ in_1^i & bool & \dots & bool & bool & \dots & bool \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ in_m^i & bool & \dots & bool & bool & \dots & bool \\ \hdashline a_1^i & bool & \dots & bool & bool & \dots & bool \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_k^i & bool & \dots & bool & bool & \dots & bool \end{array}$$

The dependencies matrix, `m_dep`, is composed of booleans which represent (1) the dependencies of the outputs of the current algorithmic iteration according to the inputs and the ageing variables of the current algorithmic iteration (2) the dependencies of the of ageing variables of the next algorithmic iteration according to the inputs and the ageing variables of the current algorithmic iteration. True expresses a dependency, false a non dependency.

- `v_in_val, v_out_val`

$$\texttt{v_in_val}^i = \begin{array}{|ccc:ccc|} in_1^i & \dots & in_m^i & a_1^i & \dots & a_k^i \\ fgdt & \dots & fgdt & fgdt & \dots & fgdt \end{array}$$

$$\texttt{v_out_val}^i = \begin{array}{|ccc:ccc|} out_1^i & \dots & out_n^i & a_1^{i+1} & \dots & a_k^{i+1} \\ fgdt & \dots & fgdt & fgdt & \dots & fgdt \end{array}$$

The input values vector, `v_in_val`, contains the values of the inputs and the ageing variables of the current algorithmic iteration. The output values vector, `v_out_val`, contains the values of the outputs of the current algorithmic iteration. The data types of theses values are fine grain data types, integer float for example, referred as $fgdt$ on the figure.
- `v_in_pre, v_out_pre`

$$\texttt{v_in_pre}^i = \begin{array}{|ccc:ccc|} in_1^i & \dots & in_m^i & a_1^i & \dots & a_k^i \\ bool & \dots & bool & bool & \dots & bool \end{array}$$

$$\texttt{v_out_pre}^i = \begin{array}{|ccc:ccc|} out_1^i & \dots & out_n^i & a_1^{i+1} & \dots & a_k^{i+1} \\ bool & \dots & bool & bool & \dots & bool \end{array}$$

The input presences vector, `v_in_pre`, is composed of booleans which represent the presences of the inputs and the ageing variables for the current algorithmic iteration. If true the corresponding input or ageing variable is available in the current algorithmic iteration. That is to say the corresponding value in `v_in_pre` is valid. The output presences vector, `v_out_pre`, is composed of boolean which indicate if the corresponding output value in `v_out_val` is valid.
- `v_out_con`

$$\begin{array}{rcccccc} & out_1^i & \dots & out_n^i & a_1^{i+1} & \dots & a_k^{i+1} \\ \texttt{v_out_con}^i = & \Big| \, bool & \dots & bool \; \vdots & bool & \dots & bool \, \Big| \end{array}$$

The output consumption vector `v_out_con` is composed of booleans which indicates which values of `v_out_val` have been consumed.

3.3.2 Methods

- `bool refresh ()` This method computes the the output and ageing variables. Its algorithm can be summarized as follow: for each column of `m_dep`, if the corresponding output or ageing variable is not computed and `v_in_pre` equals the column of `m_dep`, compute it with the expression of f_o or f_a for input output overlapping cases or f for the others cases, put the result in `v_out_val` and set the corresponding boolean to true in `v_out_pre`. For non input output overlapping cases, all booleans of m_{dep} are set to true.
- `bool in_exists (int ref)` This method is used to know if the element at row ref in `v_in_val` is already present.
- `bool out_exists (int ref)` This method is used to know if the element at row ref in `v_out_val` is valid, i.e. is computed.
- `bool put (type val, int ref)` This method is used to put the value val at row ref in `v_in_val`.
- `type get (int ref)` This method is used to get the value at row ref of `v_out_val`. When called, the boolean at row ref of `v_out_con` is set to true.
- `bool is_consumed (int ref)`
This method is used to know if the value of `v_out_val` at row ref has been consumed.
- `bool is_consumed ()` This method is used to know if all the values of the iteration have been consumed.
- `bool is_in_ageing (int ref)` This method is used to know if the value at row ref in `v_in_val` is an ageing variable.
- `bool is_out_ageing (int ref)` This method is used to know if the value at row ref in `v_out_val` is an ageing variable.

3.4 The iteration list object

The figure 4 presents the iteration object. This object is the user interface of our modelling framework. It instantiates the *iteration* and manages the inter iterations dependencies. The two following paragraphs present its attributes and methods.

```
Class iteration_list< type >

Private Attributes
        iteration< type > *     iteration_list

Public Member Functions
        bool    put (type val, int ref)
        bool    exists (int ref)
        type    get (int ref)
```

Figure 4: iteration_list object

3.4.1 Attributes

- `iteration_list`
It is an ordered list which contains the instances of alive iterations. An alive iteration is an iteration which has not consumed and produced all its fine grain inputs and outputs.

3.4.2 Methods

- `bool put (type val, int ref)` This method is used to put the input value val at position ref. Its algorithm can be summarized as follow:

1. get the older iteration which has no input value at position ref
2. if such an iteration does not exist create a new iteration and add it at the end of the list
3. put the value val at position ref in the iteration
4. refresh the iteration
5. if ageing variables has been computed put them in the next iteration. To perform that task the present algorithm is applied in a recursive way.

- `bool exists (int ref)` This method is used to check if at least one iteration contains a value at position ref.
- `type get (int ref)` This method is used to get the older value at position ref in the `iteration_list`. Its algorithm can be summarized as follow:

1. get the older iteration which has not consumed output value at position ref
2. if such an iteration does not exist return 0

3. else get the value in the found iteration
4. if this iteration is now entirely consumed, remove it from the iteration_list

3.5 Conclusion

The iteration and iteration_list objects make possible to model cases from (a) to (d), even the input output overlapping cases. However, these objects lack of a model of time. Thus, we use them in conjunction with Cofluent Studio [cfd]. The next section presents some results of this cooperation.

4 Examples

To illustrate the dependencies matrix and the fine grain synchronized algorithmic functions of F_a and F_o, the two following sub sections present these concepts applied to pedagogical, but interesting, examples: a FIR filter and a matrix product. The third sub section presents the iteration and iteration_list usage in Cofluent Studio [cfd] an Electronic System Level (ESL) design tool.

4.1 FIR Filter example

A N taps FIR filter has one input x_n, one output y_n and N coefficients h_i. It has a N-1 memory effect. Its algorithmic expression is:

$$y_n = \sum_{i=0}^{N-1} h_i x_{n-i}$$

where x_{n-i} is the value of the input i algorithmic iterations before.
Let consider the case of a four taps FIR filter. The obtained dependencies matrix is:

$\mathtt{m_dep}^i =$

	y_n^i	x_{n-1}^{i+1}	x_{n-2}^{i+1}	x_{n-3}^{i+1}
x_n^i	*true*	*true*	*false*	*false*
x_{n-1}^i	*true*	*false*	*true*	*false*
x_{n-2}^i	*true*	*false*	*false*	*true*
x_{n-3}^i	*true*	*false*	*false*	*false*

The obtained F_o set is:

$$y_n^i = h_0 x_n^i + h_1 x_{n-1}^i + h_2 x_{n-2}^i + h_3 x_{n-3}^i$$

The obtained F_a set is:

$$\begin{cases} x_{n-1}^{i+1} = x_n^i \\ x_{n-2}^{i+1} = x_{n-1}^i \\ x_{n-3}^{i+1} = x_{n-2}^i \end{cases}$$

From the analysis of M_d^i we can conclude (1) that the FIR filter computation core is able to compute the output y_n^i as soon as $x_n^i, x_{n-1}^i, x_{n-2}^i, x_{n-3}^i$ are available (2) that the ageing variables $x_{n-1}^i, x_{n-2}^i, x_{n-3}^i$ are available as soon as M_d^{i-1} is computed that is to say as soon as x_n^{i-1} arrives, and so on. The reader can point out the initialization problem: to have a working computation core, the ageing variables have an initial value in $\mathtt{v_in_val}^0$ and their corresponding boolean in the presences vector $\mathtt{v_in_pre}^0$ are set to true.

4.2 Dependencies matrix and fine grain input algorithmic expressions

The matrix product is an interesting example since its implementations have a potential fine grain input output overlapping. Moreover, a lot of DSP transforms can be written as a matrix product. Let us consider a 2x2 C=A.B matrix product. The obtained dependencies matrix is:

$\mathtt{m_dep}^i =$

	c_{11}^i	c_{12}^i	c_{21}^i	c_{22}^i
a_{11}^i	*true*	*true*	*false*	*false*
a_{12}^i	*true*	*true*	*false*	*false*
a_{21}^i	*false*	*false*	*true*	*true*
a_{22}^i	*false*	*false*	*true*	*true*
b_{11}^i	*true*	*false*	*true*	*false*
b_{12}^i	*false*	*true*	*false*	*true*
b_{21}^i	*true*	*false*	*true*	*false*
b_{22}^i	*false*	*true*	*false*	*true*

The obtained F_a set is empty. The obtained F_o set is:

$$\begin{cases} c_{11}^i = a_{11}^i b_{11}^i + a_{12}^i b_{21}^i \\ c_{12}^i = a_{11}^i b_{12}^i + a_{12}^i b_{22}^i \\ c_{21}^i = a_{21}^i b_{11}^i + a_{22}^i b_{21}^i \\ c_{22}^i = a_{21}^i b_{12}^i + a_{22}^i b_{22}^i \end{cases}$$

The analysis of the dependencies matrix and the set F_o shows that the computations of $c_{11}^i, c_{12}^i, c_{21}^i, c_{22}^i$ are now synchronized on rows of the matrix A and columns of the matrix B, for example c_{11} can be produced as soon as the first row of A and the first column of B are available. These results are well known in the case of the matrix product; nevertheless this information is of major interest in a system design refinement process.

4.3 Model introduced in an ESL design tool

In this sub-section we present how the iteration and iteration_list objects can be used in a ESL

design flow. This example relies on Cofluent Studio [cfd]. This tool aims at bridging the gap between specifications and implementation. It allows designers to model and simulate the behaviour and time properties of electronic systems applications and prepare efficiently for their implementation. Cofluent Studio allows us to add a timed behavior to our modelling framework.

To improve the readability of the time line presented below, we choose an example with a coarse grain input interface but with a smaller grain than the original input interface. Let consider the following particular refinement of a NxM matrix product C=A.B,

- it has two input ports: `a1_to_pm_refinement`, `a2_to_pm_refinement`, which carry vectors of size M. On the first one it receives the rows of the left hand side operand, on the second one it receives the columns of the right hand side operand.

- it has two output ports: `pm_refinement_01_to_c`, `pm_refinement_02_to_c`, which carry scalars. On the first one it produces the scalars of the result matrix which have an even row index, starting by the first column, on the second one, it produces the scalars of the result matrix which have an odd row index, starting by the first column.

Let $r_i, i = 0..N-1$ be the rows of the left hand side operand, let $c_i, i = 0..N-1$ be the columns of the right hand side operand, let $s_{ij}, i = 0..N-1, j = 0..N-1$ be the scalar results of the matrix product.

The following sub-sub-sections present different possible refinements of the original matrix product which obey to the structural and semantic specification given above. The first one presents a non input output overlapping case, whereas the second presents two input output overlapping cases.

4.3.1 coarse grain synchronized refinement

The coarse grain synchronized refinement consists in refining the communication interfaces without refining the original coarse grain synchronized algorithm. It refers to cases from (a) to (d) except the input output overlapping cases. Figure 5 presents a time line we obtained with Cofluent Studio [cfd] of a such possible refinement for N=3, M=5 without pipeline. The `iteration` and `iteration_list` objects are used to do the fine grain data refinement and even if this timeline does not show it, are useful to model the pipeline.

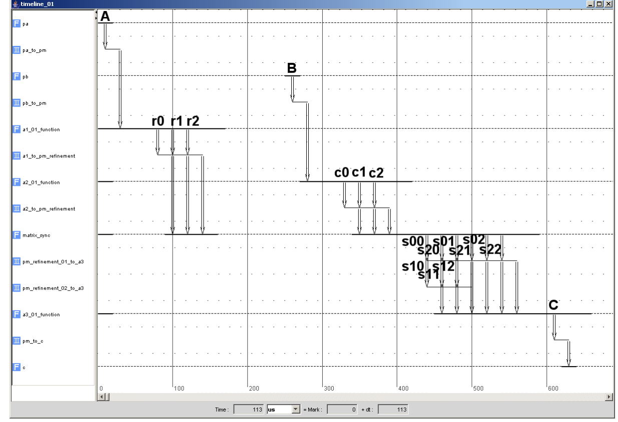

Figure 5: coarse grain synchronized refinement

4.3.2 fine grain synchronized refinement

This sub-section present two examples of input output overlapping cases. In these cases, the `refresh` method of the `iteration` object use the algorithmic expressions of F_a and F_o.

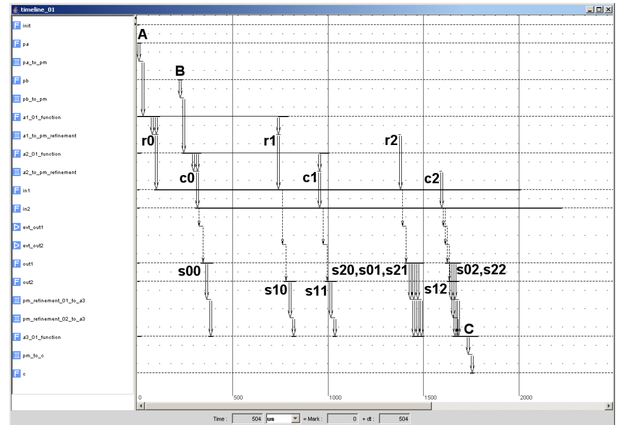

Figure 6: time line refined matrix product, reactive

Reactive model A reactive model is not scheduled and only satisfies the structural and semantic specifications. It can be inserted in a system in a top down approach to extract timing constraints [CBM03] of non scheduled components, e.g. by monitoring the input output events. In this example, if we emit the following sequence $r_0, c_0, r_1, c_1, r_2, c_2$ to this model we observe an input output overlapping, c.f. figure 6.

Fully scheduled model A fully scheduled model can be inserted in a system to model a

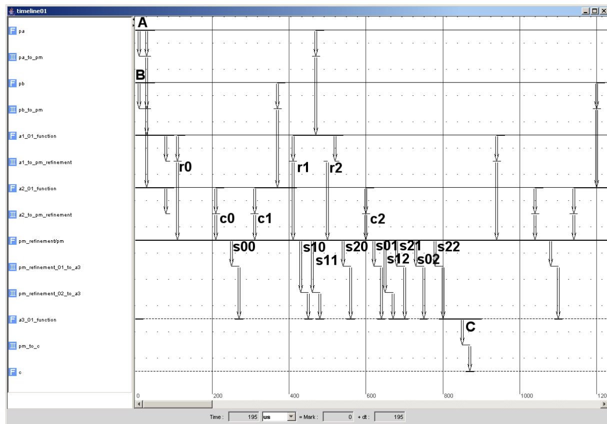

Figure 7: time line refined matrix product, fully scheduled

scheduled hardware implementation of an algorithm, it imposes timing constraints [CBM03] to the rest of the system. Its timing behaviour is fully deterministic and can be specified with Gantt diagrams. Figure 7 presents the time line of the matrix product refinement specified by the Gantt diagram of the following table.

time (us)	100	200	250	300	400	430	460	490	540	590	620	650	680	730	780
event	r0	c0	s00	c1	r1	s10	s11	r2	s20	c2	s01	s12	s21	s02	s22

5 Conclusions and Work in progress

In this paper we propose a modelling framework which can be used by a designer in a top-down refinement methodology to progressively introduce implementation details and to explore different hardware implementation alternatives, from the interfaces point of view. It can also be used in a bottom up approach to model the behaviours at the interfaces of existing hardware implementations.
We are now working on a fine grain input output constraints driven hardware implementation methodology with the Cofluent Studio tool used in conjunction with our modelling framework, in the context of software-radio applications.

6 Acronyms

DSP Digital Signal Processing

SDF Synchronous Data Flow

HLS High Level Synthesis

DFG Data Flow Graph

SFG Signal Flow Graph

FGSFG Fine Grain Signal Flow Graph

ESL Electronic System Level

References

[CBM03] Philippe Coussy, Adel Baganne, and Eric Martin. Communication and timing constraints analysis for ip design and integration. In *VLSI-SOC*, pages 38–43, 2003.

[cfd] Cofluent design, cofluent studio http://www.cofluentdesign.com.

[EH92] W. Ecker and M. Hofmeister. The design cube: a new model for vhdl designflow representation. In *EURO-DAC '92*, 1992.

[FKCM93] D. Filo, D.C. Ku, C.N. Coelho, and G. De Micheli. Interface optimization for concurrent systems under timing constraint. In *IEEE Transactions on VLSI Systems*, 1993.

[Gaj87] D.D. Gajski. The structure of a silicon compiler. In *Proceedings of IEEE ICCD*, 1987.

[gau] Lester, ubs cnrs 2734, gaut http://web.univ-ubs.fr/gaut/.

[gcc] Gnu compiler collection, http://gcc.gnu.org/.

[JKH99] Axel Jantsch, Shashi Kumar, and Ahmed Hemanimumi. The rugby model: a conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems. In *DATE '99*, 1999.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[TGCM04] F. Thabet, J.B. Le Goff, P. Coussy, and E. Martin. A methodology for timing and structural communication refinement in dsp systems. In *International Conference on Microelectronics (ICM)*, 2004.