

# Synchronous Modeling of Data-Intensive Applications

**Huafeng. Yu, A. Gamatié, É. Rutten,  
P. Boulet and J.-L. Dekeyser**

{Yu, Gamatie, Boulet, Dekeyser}@lifl.fr  
Eric.Rutten@inrialpes.fr

DART project, INRIA Futurs / WEST group, LIFL



# This talk :

- ▶ Detailed presentation of ARRAY-OL/GASPARD
- ▶ First results of study

# Plan

## Introduction

- GASPARD methodology
- General scheme

## Data-intensive processing

- ARRAY-OL language
- Existing works

## Simple synchronous modeling of GASPARD models

- Parallel model
- Serialized model

## Validation issues

## Conclusions



# Introduction

**Context** : data-intensive applications (DIA) in embedded systems

- ▶ regular multidimensional data processing
- ▶ parallel processing in System-on-Chip (SoC)

**Motivations** : adequate techniques for

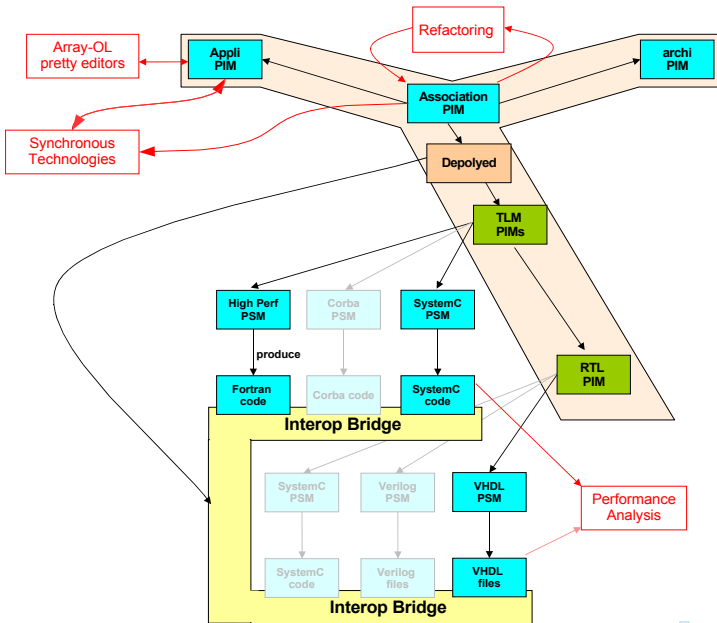
- ▶ efficient data manipulation
- ▶ analysis of implementation properties

**Approach** : combination of

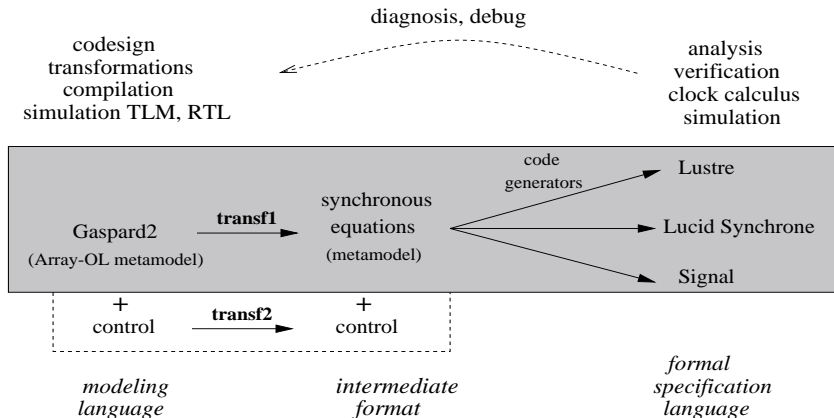
- ▶ a formalism dedicated to DIA (ARRAY-OL)
- ▶ data-flow synchronous equation models



# GASPARD methodology



# General scheme



## Introduction

- GASPARD methodology
- General scheme

## Data-intensive processing

- ARRAY-OL language
- Existing works

## Simple synchronous modeling of GASPARD models

- Parallel model
- Serialized model

## Validation issues

## Conclusions



ARRAY-OL (Array-Oriented Language) : initially proposed by Thomson Marconi Sonar [DD98]

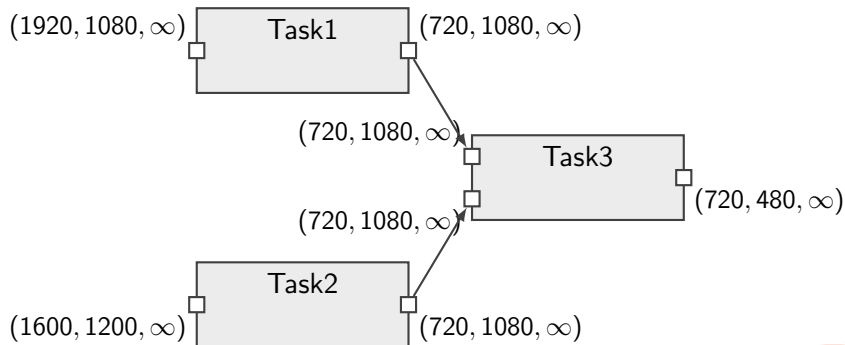
- ▶ Specification language for full parallelism
- ▶ Data manipulation through arrays
- ▶ Deadlock free and deterministic by construction

ARRAY-OL (Array-Oriented Language) : initially proposed by Thomson Marconi Sonar [DD98]

- ▶ Specification language for full parallelism
- ▶ Data manipulation through arrays
- ▶ Deadlock free and deterministic by construction
- ▶ Descriptions independent from implementation platforms
- ▶ Two types of parallelism in application specifications :  
Task parallelism and Data parallelism

# ARRAY-OL (cont'd)

## Task parallelism and data dependencies :



## Different task models :

- ▶ **Elementary task** : atomic computation block  
(instantaneous function)

## Different task models :

- ▶ **Elementary task** : atomic computation block (instantaneous function)
- ▶ **Hierarchical task** : task represented by hierarchical acyclic graphs in which
  - ▶ each node consists of a task, and
  - ▶ edges are labeled by the arrays

## Different task models :

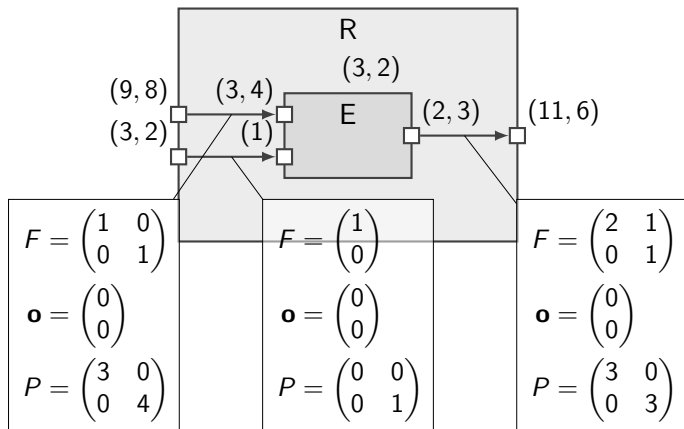
- ▶ **Elementary task** : atomic computation block (instantaneous function)
- ▶ **Hierarchical task** : task represented by hierarchical acyclic graphs in which
  - ▶ each node consists of a task, and
  - ▶ edges are labeled by the arrays
- ▶ **Repetition task** : expression of data parallelism

## Data parallelism

- ▶ **Repetition element** : the subtask to be repeated
- ▶ **Repetition space** : limitation of repetition number and link between inputs and outputs
- ▶ **Interface** : input and output arrays
- ▶ **Tiler** : defines how to obtain sub-arrays from a input array and how to store sub-arrays in a output array

# ARRAY-OL (cont'd)

## Example of a repetition task



# ARRAY-OL (cont'd)

## Tiler specification :

- ▶  $o$  : original point of the array or reference pattern
- ▶  $P$  : paving matrix (how the array is tiled by patterns)
- ▶  $F$  : fitting matrix (how patterns are filled by array elements)

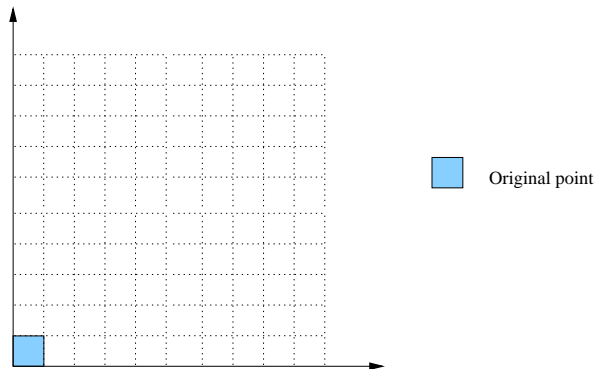


# ARRAY-OL (cont'd)

**Paving** : how the array is tiled by patterns.

These patterns are calculated in **any order**.

**Paving example** :  $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ,  $P = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$



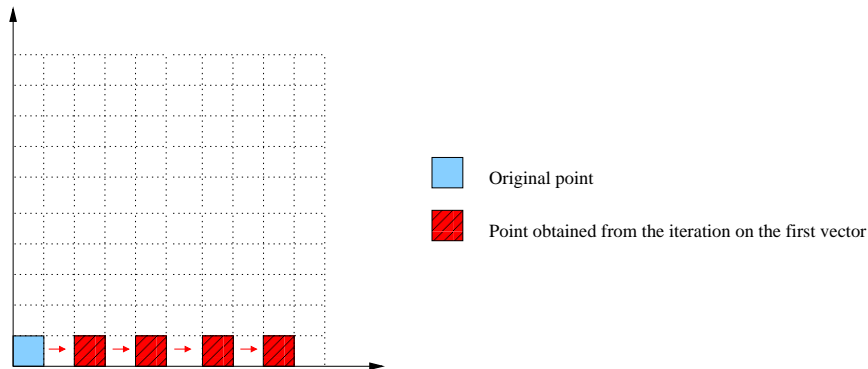
**Repetition space** :  $[5,4]$ , limitation of pattern repetitions.

# ARRAY-OL (cont'd)

**Paving** : how the array is tiled by patterns.

These patterns are calculated in **any order**.

**Paving example** :  $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ,  $P = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$



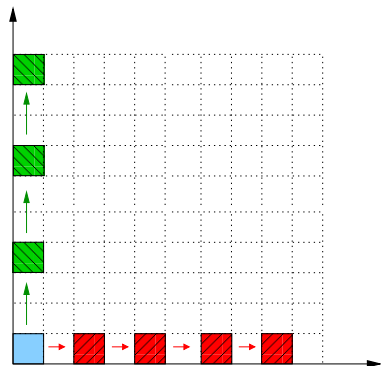
**Repetition space** :  $[5,4]$ , limitation of pattern repetitions.




# ARRAY-OL (cont'd)

**Paving** : how the array is tiled by patterns.

These patterns are calculated in **any order**.

**Paving example** :  $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ,  $P = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$



-  Original point
-  Point obtained from the iteration on the first vector
-  Point obtained from the iteration on the second vector

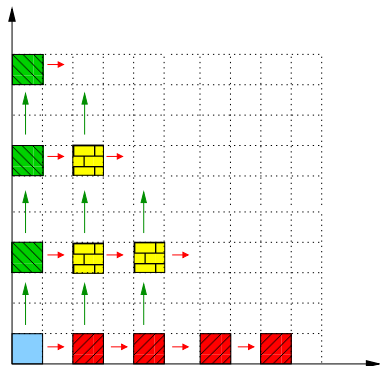
**Repetition space** :  $[5,4]$ , limitation of pattern repetitions.





# ARRAY-OL (cont'd)

**Paving** : how the array is tiled by patterns.

These patterns are calculated in **any order**.

**Paving example** :  $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ,  $P = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$



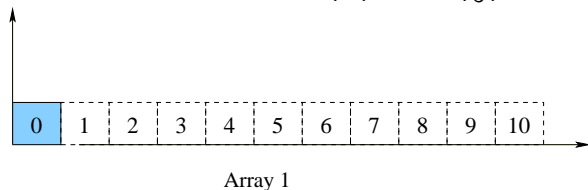
-  Original point
-  Point obtained from the iteration on the first vector
-  Point obtained from the iteration on the second vector
-  Point obtained from the iteration on the two vectors

**Repetition space** :  $[5,4]$ , limitation of pattern repetitions.

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



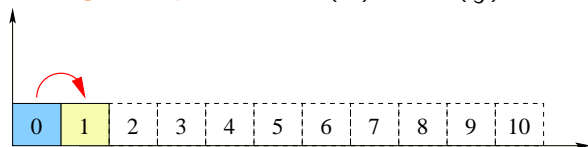
0		

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



Array 1

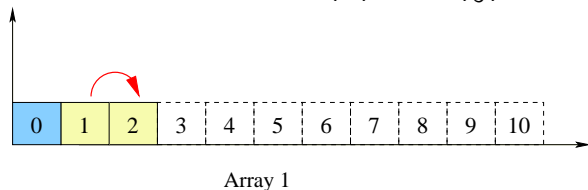
0	1	

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



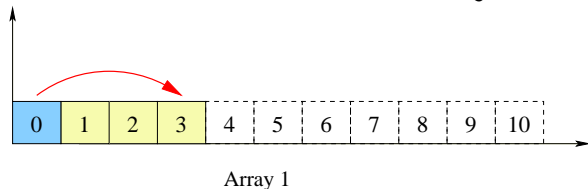
0	1	2

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



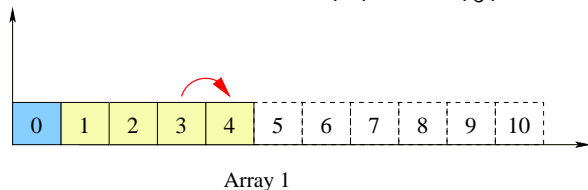
3		
0	1	2

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



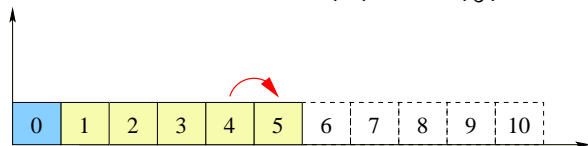
3	4	
0	1	2

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



Array 1

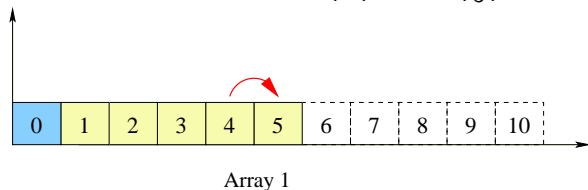
3	4	5
0	1	2

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

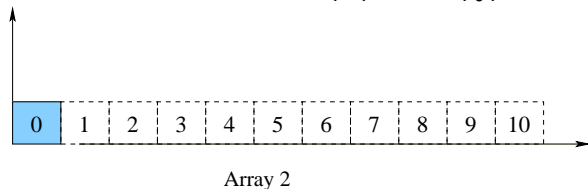
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$



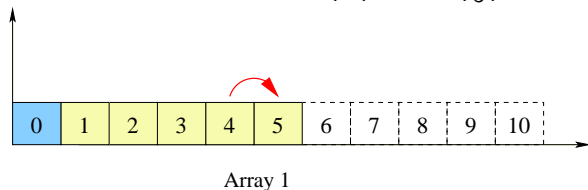
0		

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

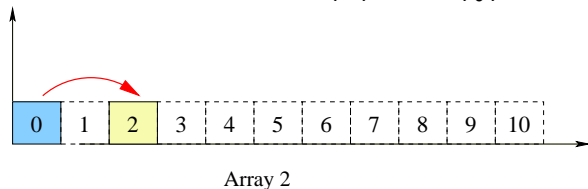
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$



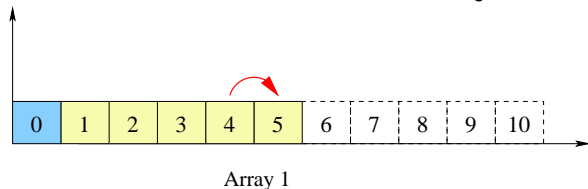
0	2	

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

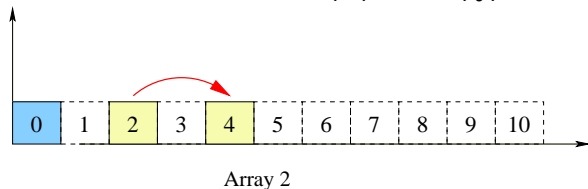
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$



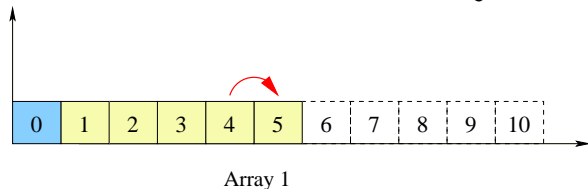
0	2	4

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

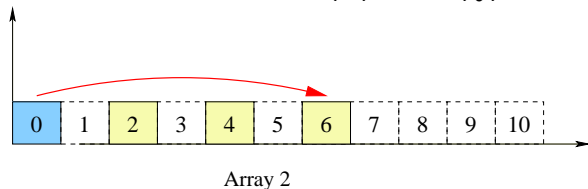
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$



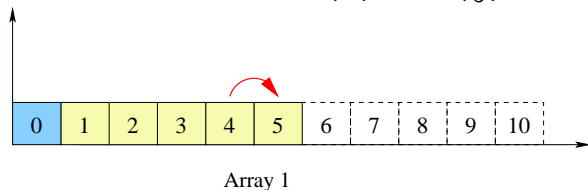
6		
0	2	4

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

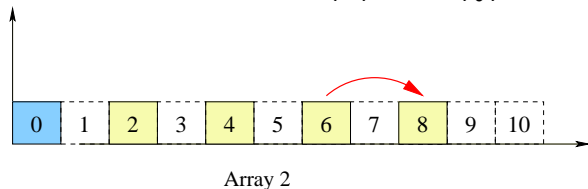
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$



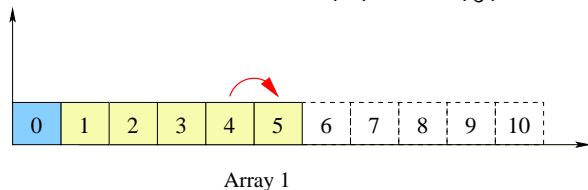
6	8	
0	2	4

Pattern

# ARRAY-OL (cont'd)

**Fitting** : how each pattern is filled by array elements.

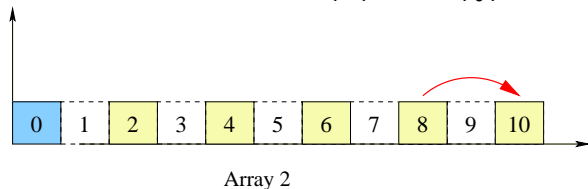
**Fitting example 1** :  $o = (0)$ ,  $F = (\frac{1}{3})$



3	4	5
0	1	2

Pattern

**Fitting example 2** :  $o = (0)$ ,  $F = (\frac{2}{6})$

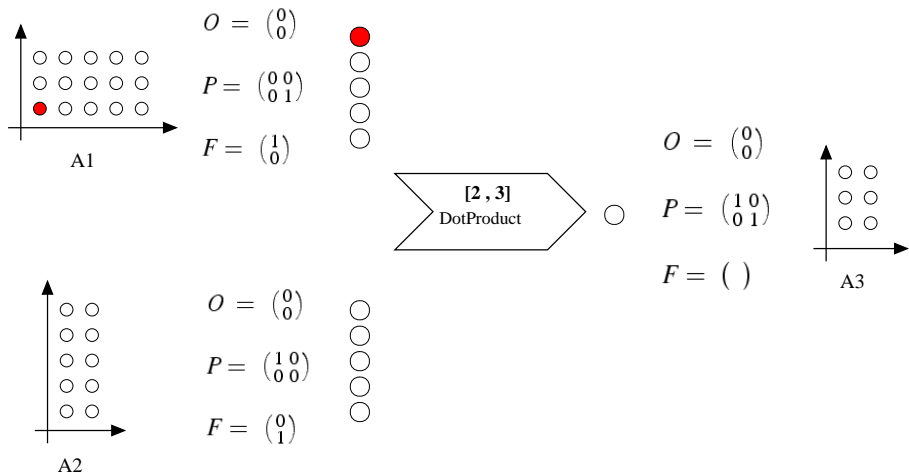


6	8	10
0	2	4

Pattern

# ARRAY-OL (cont'd)

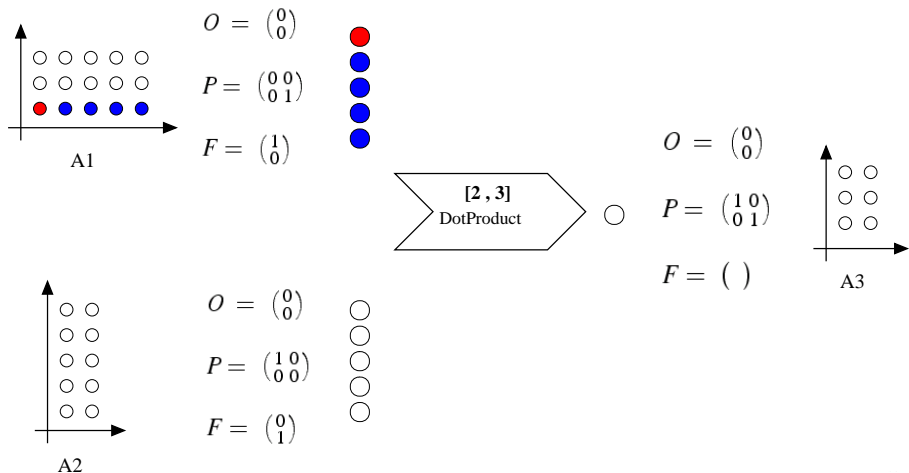
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

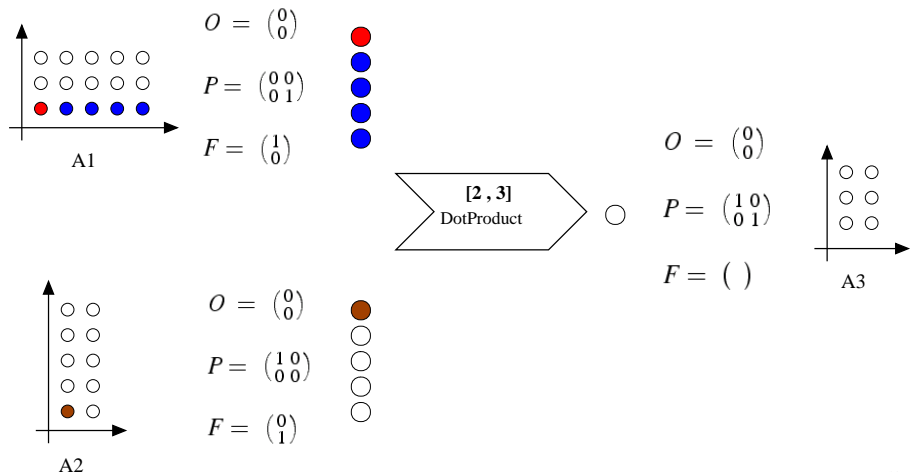
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

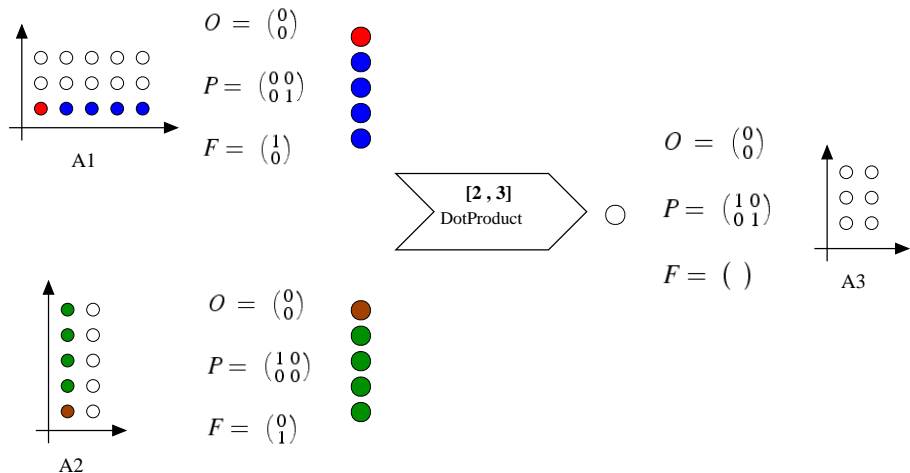
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

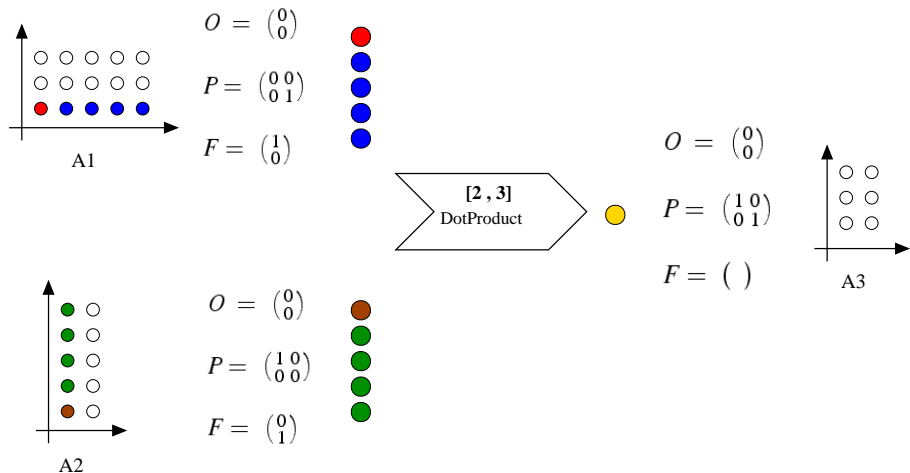
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

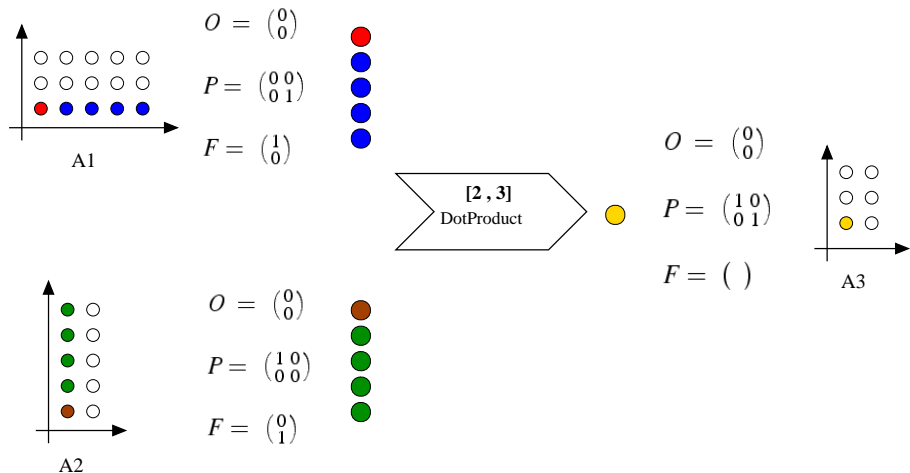
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

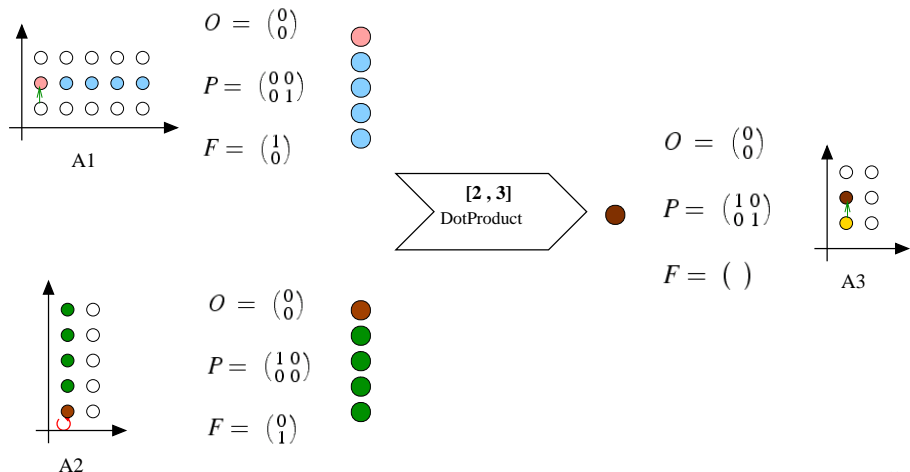
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,0]$

# ARRAY-OL (cont'd)

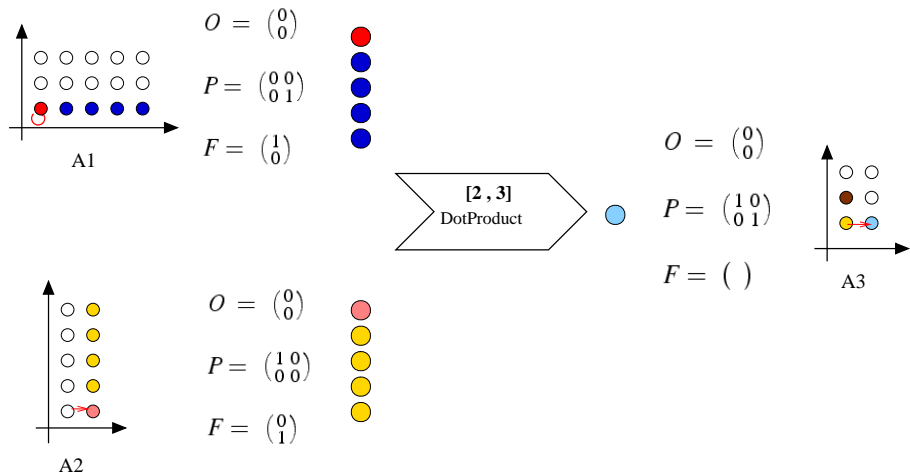
## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[0,1]$

# ARRAY-OL (cont'd)

## An example of repetition task : array product



Number of instances :  $2*3=6$  ; Repetition point :  $[1,0]$

# Existing works

**ALPHA** language [Mauras, 1989] (vs. **ARRAY-OL**)

- ▶ multidimensional data structures for data-intensive applications
- ▶ union of convex polyhedra (vs. arrays)
- ▶ data access through indices calculated by affine functions (vs. hierarchical and modular pattern )
- ▶ absence of modulo (vs. presence of modulo)



## Introduction

- GASPARD methodology
- General scheme

## Data-intensive processing

- ARRAY-OL language
- Existing works

## Simple synchronous modeling of GASPARD models

- Parallel model
- Serialized model

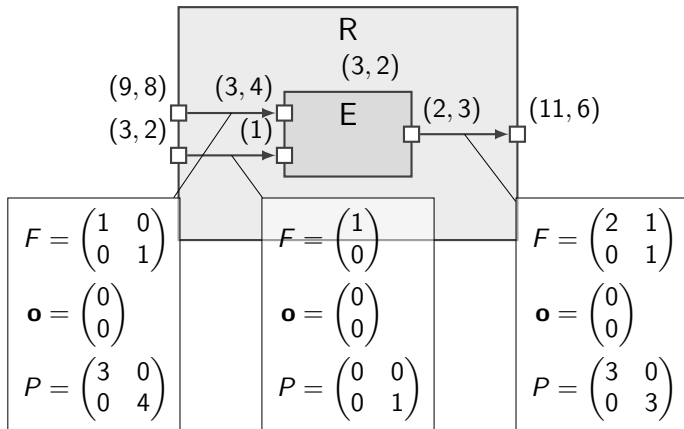
## Validation issues

## Conclusions



# Synchronous modeling of Gaspard models

Illustration of the modeling by the following example :



## Modeling of repetition task

$$\forall j \in r, A_3[\langle ind_3^j \rangle] := E(A_1[\langle ind_1^j \rangle], A_2[\langle ind_2^j \rangle])$$

- ▶  $j$  : a point in the repetition space  $r$
- ▶  $\langle ind_i^j \rangle$  : the set of index associated with pattern  $j$
- ▶  $A_i[\langle ind_i^j \rangle]$  : the pattern  $j$  associated with array  $A_i$

# Parallel model (cont'd)

## Decomposition of a repetition

**Input tilers :**  $p_1^j := A_1[\langle ind_1^j \rangle]$        $p_2^j := A_2[\langle ind_2^j \rangle]$

**Task :**       $p_3^j := E(p_1^j, p_2^j)$

**Output tiler :**  $A_3[\langle ind_3^j \rangle] := p_3^j$

Introduction of local variables :  $p_1^j, p_2^j, p_3^j$

# Parallel model (cont'd)

## Decomposition of a repetition

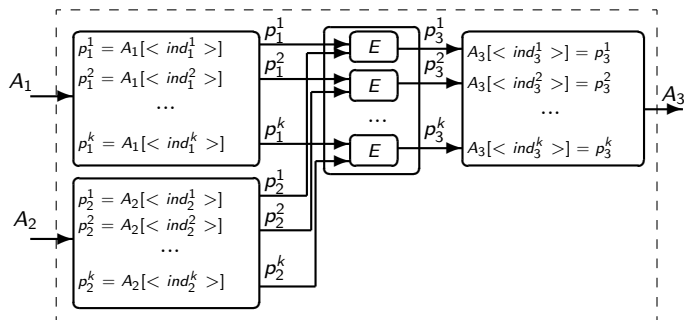
A complete system of equations :

$$\begin{array}{l} ( | p_1^1 := A_1[\langle ind_1^1 \rangle] \quad | \quad p_2^1 := A_2[\langle ind_2^1 \rangle] \\ | \quad \quad \quad p_3^1 := E(p_1^1, p_2^1) \quad | \quad A_3[\langle ind_3^1 \rangle] := p_3^1 \\ | \quad \dots \\ | p_1^k := A_1[\langle ind_1^k \rangle] \quad | \quad p_2^k := A_2[\langle ind_2^k \rangle] \\ | \quad \quad \quad p_3^k := E(p_1^k, p_2^k) \quad | \quad A_3[\langle ind_3^k \rangle] := p_3^k \\ | ) \end{array} \quad (1)$$

where  $p_1^1, p_2^1, p_3^1, \dots, p_1^k, p_2^k, p_3^k$ ; end ;

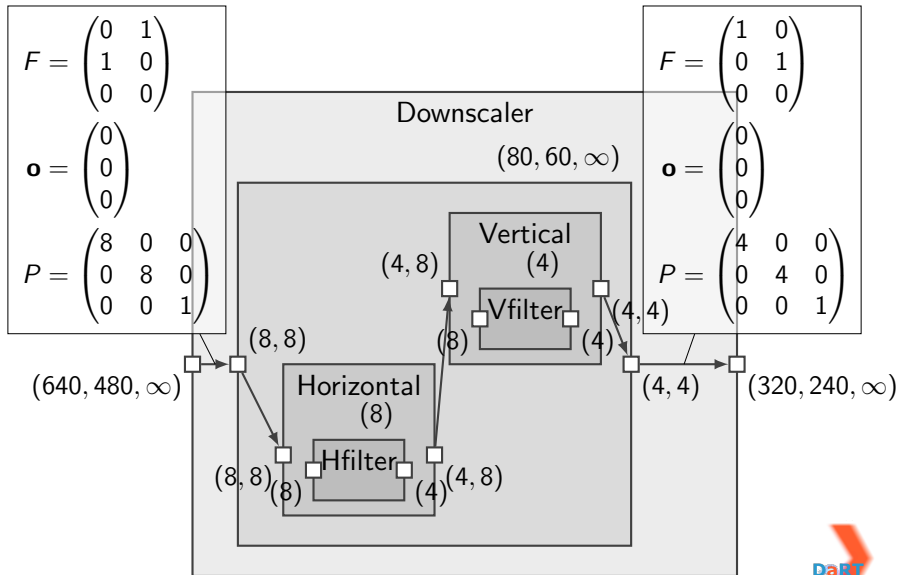
# Parallel model (cont'd)

## Restructuring and finalization of the model



- ▶ **Commutativity** and **associativity** of composition operator

# Case study : video downscaling



# Generated SIGNAL code

```
module Downscaler_module =
  process DOWNSCALER =
    (?type_array_i A_i;
     !type_array_o A_o;)
    (|(P_i1,...,P_iN:=
       HV_TILER_i(A_i)
      |(P_o1,...,P_oN):=
         R_HV_FILTER(P_i1,...
        |A_o:=HV_TILER_o(P_o1,... |)
    where
      type_pattern_i P_i1,...
      type_pattern_o P_o1,...
      process HV_TILER_i =
        (?type_array_i A_i;
         !type_pattern_i P_i1,...,
        (|P_i1:=HV_PATTERN_i1(A_i)
         |...|)
      where
        process HV_PATTERN_i1 =
```

```
      ...
      end%HV_TILER_i% ;
    process R_HV_FILTER =
      (?type_pattern_i P_i1,...,
       !type_pattern_o P_o1,...,
      (|P_o1:=HV_FILTER(P_i1)
       |...|)
      where
        process HV_FILTER =
          (? type_pattern_i P_i;
           ! type_pattern_o P_o;)
          (| p:= H_FILTER (P_i)
           | P_o := V_FILTER(p)
           |)
          where
            type_pattern_l p;
            process H_FILTER = ...
            process V_FILTER = ...
          end%HV_FILTER%;
```



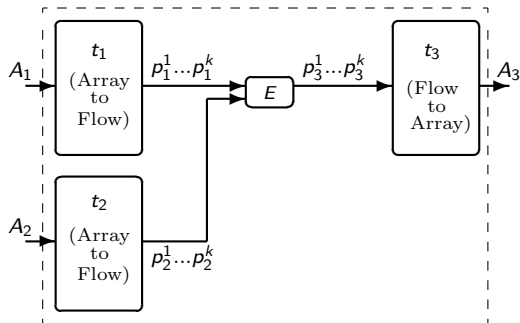
# Serialized model

- ▶ Simple parallel model
  - ▶ Semantically equivalent
  - ▶ naively enumeration
- ▶ Association of application with architecture :  
from repetition to iteration, introduction of flows
- ▶ Sequentialization at different granularity degrees [Labrani 2006]



# Serialized model

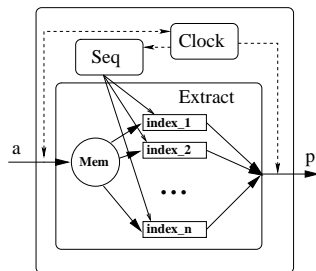
**From repetition to iteration** : introduction of flows



- ▶ **Array to flow** : produces pattern flows from arrays
- ▶ **Flow to array** : produces arrays from pattern flows

# Serialized model

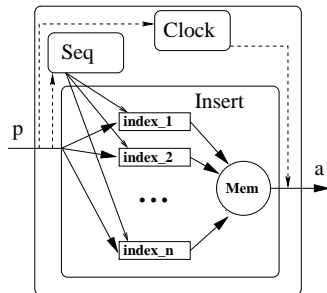
## Array to flow



Main components : clock oversampling, sequencer and Extraction

# Serialized model

## Flow to array



Main components : clock undersampling, sequencer and insertion

## Introduction

- GASPARD methodology
- General scheme

## Data-intensive processing

- ARRAY-OL language
- Existing works

## Simple synchronous modeling of GASPARD models

- Parallel model
- Serialized model

## Validation issues

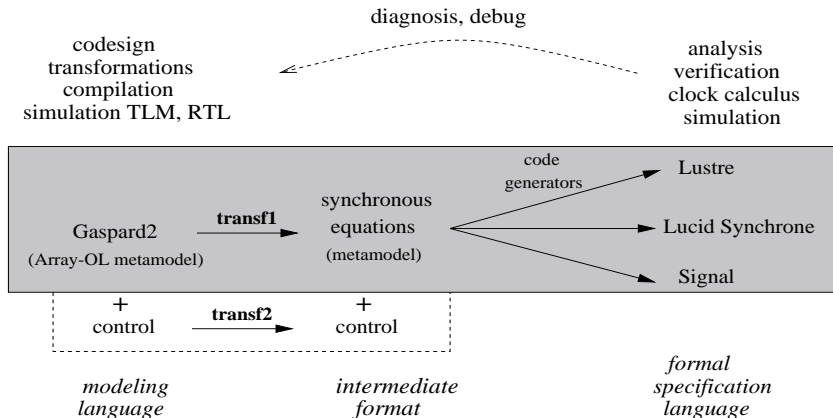
## Conclusions



# Validation issues

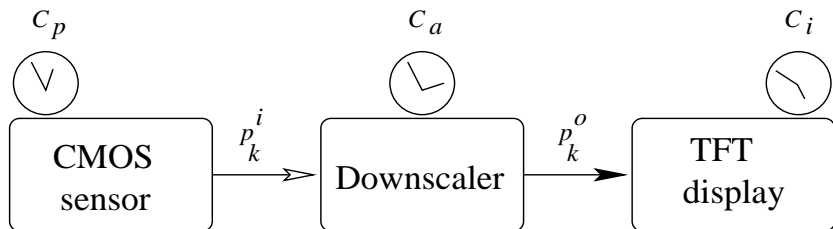
- ▶ We have a synchronous model with parallel and serialized version that can be combined (mixed model)
- ▶ We want to use synchronous analysis tools to address design correctness issues  
ex. N-synchronous Kahn network [Cohen et al. 2006], clock calculus, model-checking
- ▶ Example : a simple application with affine clocks synchronizability analysis [Smarandache et al. 1999]

# General scheme



# Synchronizability analysis

## Camera functionality in a cell phone



# Synchronizability analysis

Clock constraints :



1.  $c_a$  is an affine undersampling of  $c_p$  :  $c_p \xrightarrow{(1, \phi_1, d_1)} c_a$  ;
2.  $c_i$  is an affine undersampling of  $c_a$  :  $c_a \xrightarrow{(1, \phi_2, d_2)} c_i$  ;

# Synchronizability analysis

## Clock constraints :



1.  $c_a$  is an affine undersampling of  $c_p$  :  $c_p \xrightarrow{(1, \phi_1, d_1)} c_a$  ;
2.  $c_i$  is an affine undersampling of  $c_a$  :  $c_a \xrightarrow{(1, \phi_2, d_2)} c_i$  ;

Now, let us consider a given external constraint, which imposes a particular image production rate  $c'_i$ , from  $c_p$  such that :

$c_p \xrightarrow{(1, \phi_3, d_3)} c'_i$ . What about the **synchronizability** of  $c'_i$  and  $c_i$  ?

# Synchronizability analysis

Clock constraints :



1.  $c_a$  is an affine undersampling of  $c_p$  :  $c_p \xrightarrow{(1, \phi_1, d_1)} c_a$  ;
2.  $c_i$  is an affine undersampling of  $c_a$  :  $c_a \xrightarrow{(1, \phi_2, d_2)} c_i$  ;

Now, let us consider a given external constraint, which imposes a particular image production rate  $c'_i$ , from  $c_p$  such that :

$c_p \xrightarrow{(1, \phi_3, d_3)} c'_i$ . What about the **synchronizability** of  $c'_i$  and  $c_i$  ?

$$c'_i \text{ and } c_i \text{ are synchronizable} \Leftrightarrow \begin{cases} \phi_1 + d_1 \phi_2 = \phi_3 \\ d_1 d_2 = d_3 \end{cases} \quad (2)$$

# Conclusions and perspectives

## Current results :

- ▶ Synchronous modeling of Gaspard specifications
- ▶ Analysis of GASPARD applications with the help of synchronous techniques
- ▶ Implementation of modeling approach following MDE

## In the future :

- ▶ Complete implementation and validation of current results
- ▶ Extension with control features : mode-automata
- ▶ Using mixed models (parallel/serialized) combined with task fusion technique for placements in time and space

