Elementary transformation analysis for Array-OL

Ouassila Labbani and Paul Feautrier
École Normale Supérieure de Lyon
46 allée d'Italie 69364 Lyon, France
Email: *first name.last name*@ens-lyon.fr

Éric Lenormand and Michel Barreteau
THALES Research & Technology
RD 128 - 91767 Palaiseau, France
Email: *first name.last name*@thalesgroup.com

***Abstract*—Array-OL is a high-level specification language dedicated to the definition of multidimentional intensive signal processing applications. It allows to specify both the task parallelism and the data parallelism of these applications on focusing on their complex multidimensional data access patterns. Several tools exist for implementing an Array-OL specification as a data parallel program. While Array-OL can be used directly, it is often convenient to be able to deduce part of the specification from a sequential version of the application. This paper proposes such an analysis and examines its feasibility and its limits.**

***Index Terms*—Data parallelism, multidimensional signal processing, program analysis, Array-OL**

I. Introduction

Today embedded applications need increasing processing power, as they move from synthesis (e.g. image drawing) to analysis (e.g., face recognition). At the same time, and for basic physics reasons, technology progress is slowing down. Pending a breakthrough in non-standard technologies, the only way out is increased use of parallelism, as shown by the advent of multicore processors and Systems on Chips.

Parallel programming is admitedly difficult and error prone, hence the invention of parallel programming frameworks, where the programmer is limited to safe and easily compilable features. Examples are parallel divide and conquer, data flow (as in Kahn Process Networks) and data parallelism. Array-OL is such a framework, a combination of dataflow and data parallelism. In addition, since it was originally designed for radar and sonar applications [DG98], it is specially optimized for processing large multidimensional datasets.

Among the formalisms that target such multidimensional signal processing applications, let us quote MDSDF (Multi-Dimensional Synchronous Dataflow [ML02] and its follow-up GMDSDF (Generalized MDSDF) [ML95] proposed by Lee and Murthy. Thanks to their multidimensional structure, these models allow the expression of computation intensive signal applications. However, they do not offer effective means to exploit data or computation parallelism, and can have some constraints limiting the field of the studied applications, such as the number of dimensions which can be taken into account and mechanisms of access to the data (cyclic/acyclic, etc) [Lab06].

Another possibility is to use general purpose array processing languages, like Fortran 95 [ABM+97] or HPF [HPF94]. Parallelism in HPF is expressed by specifying the distribution of data among the processors, and then applying the "owner computes" rule. The main differences with Array-OL are, first, that HPF is intended to compile a complete application, while Array-OL tools consider the inner processing as a black box. Second, the user specified distribution of data can be quite arbitrary and complex. The generated code is difficult to optimize. In contrast, Array-OL is more constrained, which makes for more efficient implementations.

Despite the considerable research effort put presently on automating the paralellization process, it is still difficult in practice to find tools that manage automatically the entire design flow from high level source code down to executable code on potentially heterogeneous parallel computing platforms. Human involvement is still necessary, but it can (and must) be limited to high level decisions on the way the application should be partitioned and mapped on the architecture. A major advantage of Array-OL is that it provides high level, concise and human-manageable views of the data dependencies. This makes it possible to create tools like SpearDE, which allow experienced designers to control the implementation at a coarse grain level, relying on appropriate automatic tools that operate at lower level, where they are the most efficient. This approach, which first has the advantage to be operational, is efficient both in terms of productivity, as the entire final code is generated error-free, and in terms of performance as the designer can tune its mapping iteratively with rapid feed-backs from the tool on each particular mapping strategy.

In the Array-OL formalism, an application is a network of processes which communicate through shared arrays. Each process is a data-parallel program which acts repetitively on its input arrays to generate its output. Several tools and environments exist for implementing an Array-OL specification as a data parallel program such as SpearDE [LE03] (Signal Processing Environments and ARchitectures) and Gaspard2[1]. In these environments, the Array-OL formalism must be used directly. The programmer is responsible for constructing the elementary transforms, identifying the input and output regions, checking parallelism and specifying the regions parameters. To implement these tasks, a thorough knowledge of the Array-OL formalism is necessary. To avoid this problem, another possibility is to infer the Array-OL specifications from a sequential version of the program. This requires the solution of three problems:

- Rewriting the sequential program in such a way that the outer loops have no dependences.

[1] http://www.lifl.fr/west/gaspard

- Deducing the shape and size of the regions from an analysis of the array subscript functions and loop bounds.
- Rewriting the sequential code by substituting region accesses to the original array accesses.

This work is dedicated to a proposal for the solution of the second and third problems. The assumption is that one is given the sequential code, together with a list of input and output arrays, and an indication of which loop(s) are to be considered as the outer (repetition) loop(s).

II. A sketch of Array-OL

For a detailed description of Array-OL, the reader is refered to [Bou07] or [DLB+95]. As we have said earlier, Array-OL combine task parallelism – an Array-OL application is a process network – and data parallelism. In this paper, we will concentrate on the second aspect.

Data-parallelism is specified as the repetitive application of an elementary transformation to one or more input sub-arrays (*patterns*), resulting in one or more output patterns. The basic hypothesis is that all the repetitions of the elementary transform are independent and hence can be scheduled in any order, even in parallel.

At each repetition, the task reads the input pattern and performs computation to produce the output pattern. For a given input or output, all the pattern instances have the same shape, are composed of regularly spaced elements and are regularly placed in the array. Patterns are specified by the following :

- F: a fitting matrix which maps array subscripts to pattern entries
- o: the origin of the reference pattern (for the reference repetition)
- P: a paving matrix which maps repetition counters to array subscripts

For a better understanding of Array-OL concepts, consider the following elementary transformation code:

```
ET(int in[][], int out[][]){
int i,j,k, S;
Loop1: for(i=0;i<7; i++){       // Repetition loop
  Loop2 :for (k=0;k<11;k++){    // Pattern loop
    S = 0;
    Loop3: for(j=0;j<100;j++)   // Pattern loop
      S += in[0][j+11] * in[i+1][k+j];
    out[i][k] = S;
}}}
```

Here, the elementary task reads a bidimensional array `in` of unknown size to produce another bidimensional array `out`. In this simple example, the iterations of the repetition loop are independent, provided the scalar `S` is properly privatized. These iterations can be executed in any order or in parallel. The first loop `Loop1` of the elementary task is the repetition loop, while the other loops are pattern loops. At each iteration of `Loop1`, eleven elements of the output array, `out[i][0]` to `out[i][10]` are written. The pattern is of size eleven and the fitting matrix is $F = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. The pattern moves one unit in the first subscript direction at each iteration of `Loop1`, hence the paving matrix is $P = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and the origin is $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

Globally, some constraints on the number of rows and columns of the fitting and paving matrices can be derived from their use. The origin, the fitting matrix and the paving matrix have a number of rows equal to the dimension of the array; the fitting matrix has a number of columns equal to the dimension of the pattern and the paving matrix has a number of columns equal to the dimension of the repetition space.

III. Elementary transformation analysis

In the following sections, we explain how to infer pattern shape, paving and fitting matrices from a sequential code.

A. Paving

Let A be an input or output array and let its occurences in the sequential code be numbered from 1 to N. Let r be the counter(s) of the repetition loop(s), and let j^k be the counter(s) of the inner loop(s) that surround occurence k of A. Let $e^k(r, j^k)$ be its subscript function. e^k is a vector function whose dimension is the rank of A.

To be amenable to an Array-OL implementation, the subscript function e^k must be affine in r and j^k. A convenient way of checking this property consists in computing the two Jacobian matrices:

$$P^k = (\frac{\partial e^k_\alpha}{\partial r_\beta}) \quad B^k = (\frac{\partial e^k_\alpha}{\partial j^k_\beta}),$$

checking that they do not depend on r or j^k, and checking the identity:

$$e^k(r, j^k) = P^k r + B^k j^k + e^k(0, 0).$$

In Array-OL terminology, P^k is the paving matrix, and $e^k(0, 0)$ is the origin of the paving. The elements of these entities may be numbers, or they may depend on constants, which must be given numerical values just before code generation. References with different paving matrices may be separated by arbitrary distance in the source or target array; it is not possible to group them efficiently; they must be implemented as separate channels.

In the preceding example, there are two references to `in` with respective subscript functions $e^1(i, k, j) = \begin{pmatrix} 0 \\ j + 11 \end{pmatrix}$ and $e^2(i, k, j) = \begin{pmatrix} i + 1 \\ k + j \end{pmatrix}$. The corresponding paving matrices are $P^1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $P^2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Hence, the two accesses must be handled separately.

B. Pattern and fitting

When discussing patterns, one has to consider three frames of reference (see figure 1). The first one is the original (input or output) array. Its dimension is the rank of the array, noted $|A|$, and its coordinates are called *subscripts*. The shape of an array is always a (hyper-) rectangle.

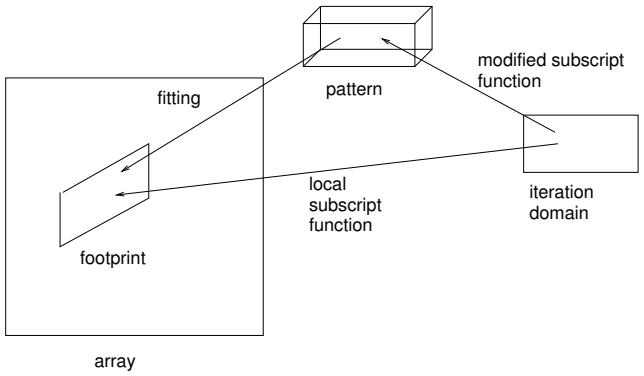

Fig. 1. Data access in Array-OL

The second frame of reference is the iteration space of the inner loops of the elementary transform. Its dimension is the number of loops enclosing the reference, noted d^k, and its coordinates are called *loop counters*. There may be as many iteration domains as there are references, or several references may share the same iteration domain. The shape of an iteration domain is arbitrary. The only requirement in the present context is to be able to construct its vertices, either because the iteration domain is rectangular, or because it can be expressed as a convex polyhedron with parameters in the constant terms only. The iteration domain of reference k will be denoted as D^k in what follows.

The third frame of reference is the pattern. According to [Bou07] the pattern is always of rectangular shape. The pattern associated to reference k is denoted by T^k and its dimension is p^k. The associated fitting matrix, F^k, connects the pattern space to the array space and its dimension, accordingly, is $|A| \times p^k$.

The relation of these objects are as follows. Firstly, the local subscript function $f^k(j^k) = B^k j^k + e^k(0,0) = e^k(0, j^k)$ gives the coordinates of an array cell relative to the reference point $P^k.r$ which moves according to the paving matrix.

Next, the image $f^k(D^k)$ is the *footprint* of reference k. Its shape is an arbitrary polyhedron. The images of the vertices of D^k by f^k form a superset of the vertices of the footprint. A polyhedron can be represented either as the set of solutions of a system of inequalities, or as the set of convex combinations of a finite number of points. There exists an efficient algorithm, the Chernikova algorithm [Sch86], for going from one of these representations to the other and back. An implementation of this algorithm is at the core of the Polylib[2].

Lastly, the image of the pattern by the fitting matrix must enclose the footprint, and it must be feasible to retrieve a datum from the pattern instead of the original array. This implies that there exists a function ϕ^k from D^k to T^k such that for every iteration vector $j^k \in D^k$, $f^k(j^k) = F^k \phi^k(j^k)$. In the text of the elementary transform, ϕ^k must be substituted to e^k in reference k to A.

As one may see from this discussion, while the iteration domain and footprint are fixed once the sequential program is given, the choice of the pattern and fitting matrix are somewhat arbitrary. There are two obvious solutions: in the first one, the pattern is the smallest rectangular box enclosing the footprint, the fitting matrix is the identity, and the subscript function is not changed. In the second solution, the pattern is isomorphic to the iteration domain (provided it is a parallelepiped), B^k is the fitting matrix, and the new subscript function is the identity.

In signal processing applications, it is often the case that several references to the same array have similar subscript functions; constructing only one pattern for several references is an interesting optimization. However, this should not be obtained at the cost of a large overhead in the size of the pattern. In other word, the number of useless elements in the pattern must be minimized. Useless elements come from two sources:

- A pattern whose shape is far from being rectangular
- A subscript matrix whose determinant is not of modulus one: there will be holes (unused elements) in the footprint. The inverse of the determinant gives an asymptotic evaluation of the ratio of useful elements.

The next section presents a method for computing a pattern and a fitting matrix in the general case (many references). This method can only be applied if all elements of the matrices B^k and the vectors b^k have known numerical values. Section III-B2 presents fail-soft solutions for cases in which these elements depend on unknown parameters.

1) The General Case: The basic observation is that a conservative estimate of the footprint can be obtained by computing the projection of each iteration domain by the associated subscript function, then constructing a convenient superset of the union of these projections. One practical method consists in projecting the vertices of the iteration domains. One then gathers all such projections, and constructs their convex hull by familiar (e.g., Chernikova's) algorithms.

To reduce the size overhead, one should notice that a useful point for reference k also belongs to the lattice which is generated by the column vectors of B^k. Hence, B^k, properly simplified (see later) could be used as the fitting matrix. However, in the case of several references, we have to combine several lattices into one, since each pattern has only one fitting matrix. As an illustration of this construction, consider the one-dimensional case. A one-dimensional lattice is simply a set of regularly spaced points. Combining two lattices generates a lattice whose spacing is the greatest common divisor (gcd) of the component spacings. The many-dimensional equivalent of the gcd is the construction of the Hermite normal form of the subscript matrices.

Let $\Lambda(B, b)$ be the lattice generated by B with origin b, i.e. the set of points $\{Bx + b \mid x \in \mathbb{N}^d\}$. Let $L^1 = \Lambda(B^1, b^1)$ and $L^2 = \Lambda(B^2, b^2)$ be two such lattices. It is easy to see that the union of L^1 and L^2 is included in the lattice $L = \Lambda([B^1 B^2 (b^2 - b^1)], b^1)$. This construction can be extended to any number of component lattices. The resulting matrix is $[B^1 \dots B^N (b^2 - b^1) \dots (b^N - b^1)]$ and the origin is b^1. Furthermore, b^1 can be moved to the origin of the paving and

[2] `http://icps.u-strasbg.fr/polylib`

hence taken as 0 when computing the fitting. However, this matrix is highly redundant. It can be simplified by using the well known fact that the Hermite normal form of B generates the same lattice as B [Sch86].

It is interesting to notice that this general solution reduces to one of the approximate methods above in special cases. If B is unitary, then its Hermite normal form is the unit matrix. In that case, the pattern is the footprint, eventually extended to a rectangular box and the fitting matrix is the identity. Conversely, if B is already in Hermite normal form, the pattern is isomorphic to the iteration space, and B is the fitting matrix.

2) The Parametric Case: Parameters occurs mostly in loop bounds. They may also appear as strides and, more seldom, in the coefficients of subscript functions.

In the Array-OL formalism, the repetition loops must be square. Hence, their bounds may be extracted directly from the program text. The extraction of the paving matrix is a simple derivative computation, which is an easy task for a competent computer algebra system. Similarly, the B^k matrices are the result of a derivation, and may contain parameters.

There are no restrictions on the inner loops. For the construction of the pattern, one needs to know the vertices of the inner iteration domain. There are three cases:

- The bounds are constant: they can be extracted even if parametric.
- The bounds are affine expressions in outer loop counters and parameters: the vertices can be computed with the help of Chernikova's algorithm, as implemented for instance in the Polylib.
- In other cases, there is no way of computing vertices, but the user may supply a bounding box.

The computation of the Hermite normal form can be done only if the matrix is known numerically, except in two cases: the matrix is 1×1 (it is its own normal form) or 2×2.

If none of these circumstances applies, the solution of last resort is to use one of the approximate schemes above. For instance, if the vertices of the inner iteration domain are available, it is possible, whatever the B matrix, to compute the vertices of the footprints and to enclose them in a rectangular box. The paving matrix is then the identity.

3) Degenerate cases: Degenerate cases occur in the following situations:

- Access to scalars or constant subscripts
- No repetition loops
- No pattern loops

These cases have to be handled by specific code since the underlying computer algebra system has difficulties understanding matrices with no rows or no columns. Let us consider the case when there are no pattern loops. One can still compute a fitting matrix for each access and group accesses according to their paving matrix.

In a group, accesses may differ by the constant part of the subscript function. In that case, one may still compute a fitting matrix – via the Hermite normal form construction – and a bounding box. If there is only one access, or if all accesses have the same constant part, the pattern has only one element, and, by convention, the fitting matrix is empty.

It is also possible that the studied application does not respect the Array-OL model semantics in the following situations:

- Several references to the same input or output array with different paving
- Several repetition loops at the same level
- Sequential loops with recurrence on a given variable
- Array used as subscript

In these cases, it is difficult, even impossible to compute the Array-OL informations and predict the pattern shape.

As discussed in section III-A, if there are several references to the same input or output array, our general algorithm computes one paving matrix for each reference. This is not allowed in the Array-OL model if the paving matrices are different. In this model, the different accesses to the same array must be handled separately. A possible solution is then to implement the array accesses as separate channels.

Using different repetition loops at the same level is also not allowed by the Array-OL model. In this model, each repetition loop nest must be performed as a separate elementary transform. To solve this problem, we can compute the pattern for each repetition loop nest, and then join the different resulting patterns.

If the studied application contains loops that can not be executed in parallel, it is difficult to model it in the Array-OL parallel model as explained by the example of section V. It is also impossible to predict the pattern and Array-OL parallel informations if the subscript function contains array since we have no information on the content of this array as in the following example:

```
int tab_inter[];
int IN1_D1,IN1_D2, IN1_D3, OUT1_D1;
void SP_Interleave(
     int D_In[IN1_D1][IN1_D2][IN1_D3],
     int D_Out[OUT1_D1]){
  int bloc,symb,b;
  int Ncbps, nb_sous_p_utiles;
  int Dim1_bits_poinconnes;
  Dim1_bits_poinconnes = OUT1_D1;
  for(bloc=0;bloc<Dim1_bits_poinconnes;bloc++){
    for(symb=0;symb<nb_sous_p_utiles;symb++) {
      for(b=0;b<nb_bits_par_symb;b++) {
        D_In[bloc][symb][b] =
        D_Out[tab_inter[symb*nb_bits_par_symb+b]];
      }
    }
  }
}
```

In the present implementation, and in order to simplify the processing of the special cases which are not allowed by the Array-OL model, we decided that the paving matrix is empty, the fitting matrix is the identity and the pattern is the whole array. A warning message is also displayed to inform the user.

IV. Implementation

A prototype tool has been implemented as an extension to the Syntol scheduler [Fea06]. CRP, the input language of Syn-

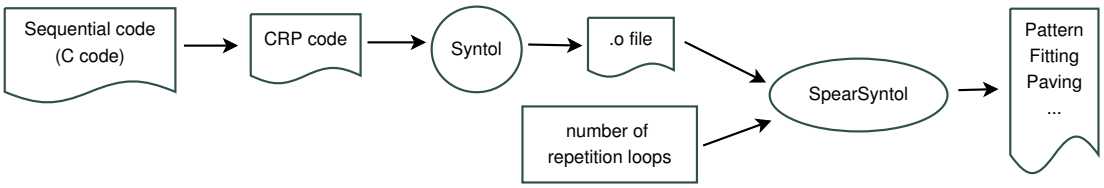

Fig. 2. SpearSyntol implementation

tol, is a specification language for processes communicating via shared arrays. The basic syntax of CRP is ANSI C. The conversion of ordinary C code to CRP is straightforward and completely automated.

The algorithms we have presented in the preceding sections have been implemented as an additional pass in Syntol. Syntol is responsible for syntax analysis and checking, loop identification and array accesses extraction. As a side effect, Syntol catches many errors and inconsistencies, like type and subscript errors. The output of Syntol is submitted to the SpearSyntol pass, which has been implemented within the MuPAD[3] computer algebra system. SpearSyntol needs another information: the number of repetition loops, which cannot be inferred from the program text.

The first step in this analyser is the detection of the repetition loops and the pattern loops. Thereafter, the analyzer computes for each input and output array the pattern shape, the origin reference and the paving and fitting matrices.

The results of this analysis can be given in several forms: a simple output log or an XML file or as additions to an Eclipse Ecore model. A gateway from SpearSyntol to SpearDE has also been developped. Figure 2 is a flow diagram of the the SpearSyntol analyzer.

V. Case study

As a case study, we have used a prototype implementation of a full 802-16 modem transmit and receive processing chain as implemented at Thales on SpearDE from a hand-made description of the elementary transformations (ETs). A random sequence of bits is encoded, interleaved, and converted to a sequence of signal samples that are passed through a radio channel simulator. The output of this simulator is submitted to the inverse processing chain, and the decoded sequence is compared to the source. The corresponding 23 ETs represent about 1200 lines of C (see Figure 3). An example of ET code is given below:

```
void SP_Collapse_depunc(
     double D_In[SP_Collapse_depunc_IN1_D1]
                [SP_Collapse_depunc_IN1_D2],
     double D_Out[SP_Collapse_depunc_OUT1_D1]){
  int Dim1_In,  Dim2_In;
  int ofsM, bdM, b,ofs;
  Dim1_In = SP_Collapse_depunc_IN1_D1;
  Dim2_In = SP_Collapse_depunc_IN1_D2;
  ofsM = Dim1_In;
  bdM = Dim2_In;
  for(ofs=0;ofs<ofsM; ofs++)
    for(b=0;b<bdM; b++)
      D_Out[ b+ ofs*bdM] = D_In[b][ofs];
  SP_Collapse_depunc_OUT1_D1 = bdM*ofsM;
}
```

[3]http://www.mupad.de

Arrays appear in the ET's declaration with their dimension (1 or 2 here), and the type of their data (int, double, ..). Their extent in each dimension is given as a parameter which is reused within the ET, in particular to derive the loop bounds. In the above example, the main loop on `ofs` is declared elsewhere as being the repetition loop, i.e. the associated Jacobian will be used to create the paving matrices (stride of 1 on the first dimensions of both input and output). The internal loop on b is analysed to find a fitting of 1 on the second dimension of the input array and a pattern length equal to parameter `bdM`.

The next example illustrates the case where an array is referenced several times:

```
int poincon;
int SP_Puncture_IN1_D1, SP_Puncture_OUT1_D1;
void SP_Puncture(
     int D_bits_codes[SP_Puncture_IN1_D1] ,
     int  D_bits_poinconnes[SP_Puncture_OUT1_D1]){
  int i;
  int Dim1_bits_codes = SP_Puncture_IN1_D1;
  Dim1_bits_codes = SP_Puncture_IN1_D1;
  for (i=1; i<=(Dim1_bits_codes/4); i++){
    D_bits_poinconnes[(i-1)*3 + 0]  =
                     D_bits_codes[(i-1)*4 + 0];
    D_bits_poinconnes[(i-1)*3 + 1]  =
                     D_bits_codes[(i-1)*4 + 1];
    D_bits_poinconnes[(i-1)*3 + 2]  =
                     D_bits_codes[(i-1)*4 + 3];
  }
  SP_Puncture_OUT1_D1 = 3*Dim1_bits_codes/4;
}
```

In that case, the analyser computes for each array a rectangular approximation of the pattern; namely of length 4 for the input and 3 for the output. Most elementary transforms in this experiment were correctly analyzed by our tools. Some of the difficulties came from genuine algorithmics problems. For instance, in the bit decoding routine, one find a loop:

```
  current = 0;
  for(i=0; i<n; i++){
      current = In[current][n-i];
      Out[n-i] = etat & mask;
  }
```

This loop cannot be executed in parallel, since there is a recurrence on `current`. Furthermore, since this variable takes its values from the array `In`, which is an input to the routine, it is impossible to predict its value as a function of the iteration variable. Hence, the indexing pattern of `In` is unknown at compile time. The only possibility is to decide

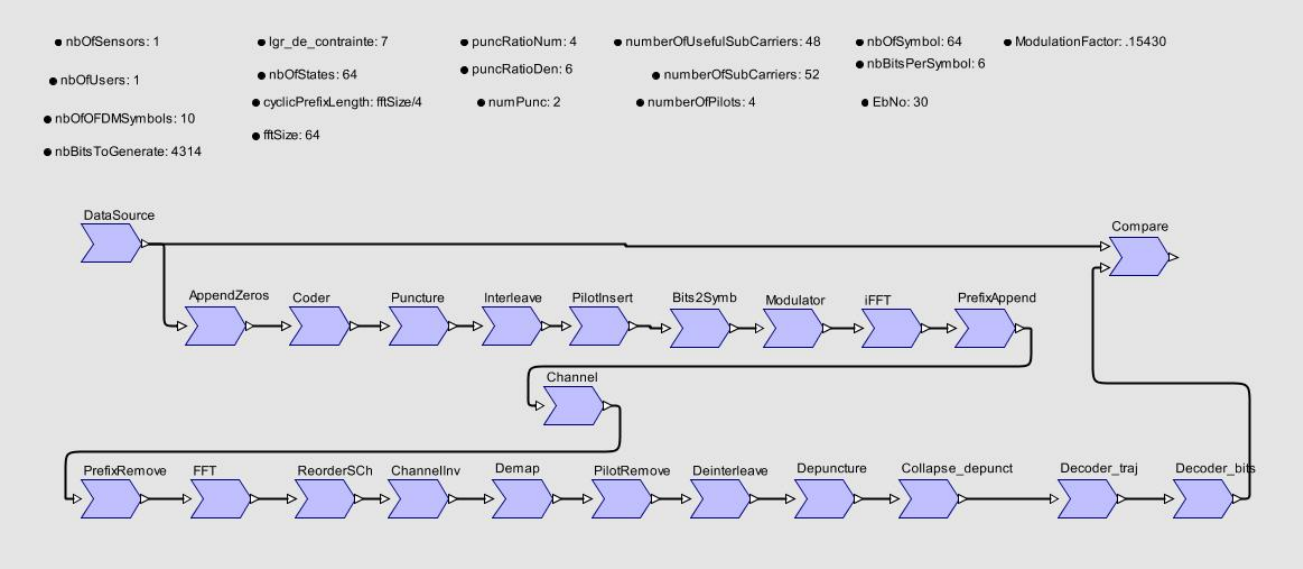

Fig. 3. Elementary transformations in the modem example

that `In` has no paving, and that its fitting is the identity, the pattern being the whole array.

Other difficulties come from the coding practices of the writers of elementary transforms. In many cases, array are not declared as such, but accessed through descriptors. Loop bounds are computed from information in these descriptors. Since many array dimensions are not given, it may be necessary to infer them from the loop bounds, by a process akin to symbolic execution. In the exemple above, there is no information in the program text about the first dimension of `In`; it must be supplied by the user.

VI. Conclusion and future work

In this paper, we have explained how to analyse elementary transformation to infer Array-OL specifications. The objective of this work is to facilitate the use of Array-OL formalism and to allow the re-use of legacy code in a parallel context. The Array-OL system is dedicated to the specification of intensive signal processing applications, and its main characteristic is that it allows the exploitation of the full parallelism in these applications.

In the future, we will study more general cases, in particular the possibility of manipulating skewed patterns. It is also possible to benefit from Syntol results for a better analysis of the parallel application. Since the Syntol tool computes dependences, it is thus possible to check that the repetition loops are actually parallel. One must take care that Syntol will find dependences if temporary scalars are used in the code of the elementary transforms. These scalars must be expanded or privatized at code generation time.

Overlap between patterns (or, rather, between footprints) is another concern. For input arrays, overlap is just a cause of inefficiency, since some arrays cells will be copied several times to processors. Overlap for output arrays are more dangerous since they may induce non-determinism. The existence of overlap may be tested provided one stays inside the polytope model (affine loop bounds and indexing functions, with numerical coefficients and linear parameters).

References

[ABM+97] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Jerrold L. Wagener and Brian T. Smith. *Fortran 95 Handbook: Complete ISO/ANSI Reference*. MIT Press, Cambridge, 1997.

[Bou07] Pierre Boulet. *Array-ol revisited, multidemensional intensive signal processing specification*. Research Report RR-6113, INRIA, February 2007.

[DG98] Alain Demeure and Yannick Del Gallo. *An Array Approach for Signal Processing Design*. In Sophia-Antipolis conference on Micro-Electronics (SAME 98), France, October 1998.

[DLB+95] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. *Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel*. In *Gretsi* (Groupe d'Etudes du Traitement du Signal et des Images), pages 1029-1032, France, 1995.

[Fea06] Paul Feautrier. *Scalable and Structured Scheduling*. International Journal of Parallel Programming, 34(5), pages 459-487, Norwell, MA, USA, 2006.

[HPF94] CORPORATE High Performance Fortran Forum. High performance Fortran language specification (part III). SIGPLAN Fortran Forum, 13(3), pages 22-55, ACM, New York, NY, USA, 1994.

[Lab06] Ouassila Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. PhD thesis, LIFL, Université des Sciences et Technologies de Lille, November 2006.

[LE03] Eric Lenormand and Gilbert Edelin. *An Industrial Perspective: A pragmatic high-end signal processing environment at Thales*. In SAMOS: 3rd international workshop on synthesis, architectures, modeling and simulation, pages 52-57, 2003.

[ML95] Praveen K. Murthy and Edward A. Lee. *A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices*. Proceedings of the ICASSP'96, Atlanta GA, May 1996.

[ML02] Praveen K. Murthy and Edward A. Lee. *Multidimensional synchronous dataflow*. IEEE Transactions on Signal Processing, 50(8):2064-2079, august 2002.

[Sch86] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, NewYork, June 1998.