Hierarchical Static Analysis of Structured Systems of Affine Recurrence Equations

Florent de Dinechin[*], Sophie Robert[†]
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

The Alpha *language, based on systems of affine recurrence equations over polyhedral domains, allows the expression of complex algorithms as hierarchical, parameterized structures of such systems. This paper discusses the static analysis of* Alpha *programs, an extended type-checking process based on the single assignment rule. We present techniques ensuring, on one hand, that a system is valid (with respect to this rule) within a certain range of its parameters, and on the other, that no system is used with invalid values of its parameters. These techniques have been implemented in a versatile tool for* Alpha *which is demonstrated on the example of the Gaussian elimination.*

1: Introduction

The background of this paper is the formalism of *recurrence equations* first introduced by Karp, Miller and Winograd [6]. A regular problem or algorithm, expressed at the highest level as a *system of affine recurrence equations* (SARE), may be transformed, through a range of (semi-)automatic, provably correct, *refinement* steps, into an implementation-level description such as a VLSI regular array [5], imperative code [10], or a collection of loop nests aimed at a massively parallel architecture [9]. This paper addresses more specifically the *polyhedral* model, in which the SARE variables are data arrays with polyhedral domains. The results presented here have been implemented in the Alpha environment [8, 12], which is based on polyhedral SAREs [13].

One of the strong points of data-parallel models [4] is the possibility of *statically* analyzing algorithms to extract the information (e.g. dependency information) needed by transformation tools. In this paper we show how such a static analysis may also be useful at the early development stages: we show that there is a large set of conditions on the correctness of a program which may be checked statically by an automatic tool, before any simulation step. Due to lack of space, we concentrate here on static checks based on the denotational semantics of the language [8]. Other very important checks, which are more classical and purely syntactic (e.g. type-checking) are not described here, although also implemented.

Basically, the semantic static analysis reduces to verifying the *single assignment rule*, i.e. ensuring that the variables are correctly defined wherever expected. This analysis is done in a formal manner, using symbolic computations over the polyhedra. One might think that similar results could be obtained by an exhaustive enumeration of the domains, however our approach has several advantages:

[*]Partially supported by a grant from France Telecom (Cairn Project)
[†]Supported by the Portrait Project, European Copernicus Scientific Program CP940682

- It is possible to analyze SAREs with infinite domains (e.g. a convolution involves infinite data streams).
- It is possible to analyze parameterized SAREs. The range of the parameters being often infinite, an exhaustive enumeration would be impossible.
- It is possible to detect problems specific to particular values of the parameters.
- Error messages are more verbose and synthetic, and may be parameterized, too.
- Last but not least, it is faster and independent of the size of the domains (although it depends on their dimensionality).

However, even with such a static analysis tool, it is difficult to develop complex (real-world) algorithms as single SAREs, in particular because of their high dimensionality. For instance, a *singular value decomposition* (SVD) [1, 2, 7], involves data arrays of dimension 5. The *SARE structures* of the ALPHA language [3] were designed to address the need for *divide and conquer* in terms of dimensionality. They enrich the language with the usual decomposition of a program into functions, but also with more complex program structures which may be interpreted as function calls within loop nests in imperative languages, or as an extension of the functional *map* operator to affine polyhedra instead of lists. Thus a complex algorithm like the SVD may be broken down into smaller systems of increasing dimensionality, just as the deep loop nest, in the sequential algorithm, is broken down into functions consisting of simple loops. This paper shows that this *divide and conquer* strategy is also valid for the analysis of large programs expressed as hierarchies of parameterized SAREs: we show how the static analysis tool encourages the user to restrict the parameters of each SARE to a maximal range where the SARE is valid. Then this SARE may be used in a context of greater dimensionality, so its parameters are assigned a value which may vary in an affine manner. We show how the tools checks, in this case, that the actual parameters assigned fall within this parameter domain, which may therefore be viewed as a set of *preconditions* on the parameters. The fact that these preconditions may only be affine ones may seem very restrictive, but surprisingly much can be expressed this way since we only deal with regular algorithms.

The paper is organized as follows. Sect. 2 briefly introduces the basics of the polyhedral SARE model, with the ALPHA syntax which we will use in the following. Sect. 3 presents the analysis of a single, parameterized SARE. Sect. 4 discusses the hierarchical analysis of a structure of SAREs.

Examples all along are based on a simple, two-system ALPHA program for the Gaussian elimination (Fig. 1), and the checks performed will therefore seem trivial. Most of this work, however, was done while trying to implement two SVD algorithms, leading to programs several pages long with 8-dimensional computation spaces (5 apparent dimensions and 3 size parameters) which happened to simulate correctly as soon as they passed our static analysis.

2: Basics

This section presents the formalism of polyhedral SAREs as it is implemented in ALPHA. The example program (Fig. 1) will illustrate most of the concepts presented here, and the interested reader is refered to [8, 3, 12] for more details.

Recurrence equations relate *spatial variables*, which are arrays of values defined over some

```
1    system ZeroColumn: {N,K| 1<=K<N} (A : {i,j | 1<=i,j<=N} of real )
2                              returns (Ar: {i,j | 1<=i,j<=N} of real );
3    let
4       Ar[i,j] = case
5                    {| i<=K}      : A[i,j];
6                    {| i>K; j<=K} : 0[];
7                    {| i>K; j>K}  : A[i,j] - A[K,j]*A[i,K]/A[K,K];
8                  esac;
9    tel;
10
11
12    system Gauss: {N | N>1} (A : {i,j | 1<=i,j<=N} of real)
13                     returns (T : {i,j | 1<=i,j<=N} of real);
14    var  Ak : {i,j,k | 1<=i,j<=N; 1<=k<=N} of real;
15         Ak1: {i,j,k | 1<=i,j<=N; 1<=k<N} of real;
16    let
17       use {k| 1<=k<N} ZeroColumn[N,k] (Ak) returns (Ak1);
18       Ak[i,j,k] = case
19                      {| k=1} : A[i,j];
20                      {| k>1} : Ak1[i,j,k-1];
21                    esac;
22      T[i,j] = Ak[i,j,N];
23    tel;
```

Figure 1. Gaussian elimination in Alpha

index space (or *domain*). In the polyhedral model, these domains are *convex polyhedra* of some integer vector space $\mathbb{Z}^n$. A convex polyhedron may be described as a finite intersection of *halfspaces* expressed as affine inequations of the coordinates in $\mathbb{Z}^n$. An example of a polyhedron with the ALPHA syntax is `{i,j| 0<=i,j<N; i<=j}`. The strong point of this model is that the set *DOM* of finite unions of such integral convex polyhedra has a lattice structure under inclusion, and is closed under the following operations: intersection, union, set difference, preimage by an affine function, and convex hull of the image by an affine function. This allows the easy formal manipulation of such domains [13].

An ARE defines the value of such a spatial variable at each point of its domain as a function of other values. This function is given as an expression relating the variables and involving computation (or pointwise) operators, and communication (or spatial) ones. These operators are detailed later in 3.1. The dependency of one variable to another must be affine. Thus an ARE expression, like a variable, denotes a polyhedral data array, with a domain and values which are a combination of those of the variables.

A SARE defines the values of some *output* variables as a function of the values of some *input* variables, using mutually recursive AREs involving the input, output and *local* variables. The class of each variable, along with the type of its values, is declared in the header of the system. For instance, in system `Gauss`, `A` is input, `B` is output, `Ak` and `Ak1` are local.

Furthermore, an ALPHA SARE may be parameterized by any number of *size parameters* whose permitted range is given as a *parameter domain* in the header of the system. These

parameters are actually implicit indices of all the objects (domains and affine functions) appearing in the system. For example the system `Gauss` of Fig. 1 has `{N| N>1}` as a parameter domain. Therefore, within this system, the two-dimensional parametric domain `{i,j| 1<=i,j<=N}`, which is the domain of the input variable `A`, actually denotes the following 3-dimensional closed domain: `{i,j,N| 1<=i<=N; 1<=j<=N; N>1}`. Thus parameterized domain also belong to *DOM*.

Lastly, it is possible to define hierarchical structures of parametric SAREs thanks to the **use**[1] construct [3], which allows us to increase the dimensionality of a subsystem by instantiating a regular collection of this subsystem. As an example, the line containing the **use**, line 17 of Fig. 1, should be read: instantiate a regular collection (indexed by `{k|1<=k<N}`) of system `ZeroColumn`, assigning the values `N` and `k` to the parameters of instance `k`, the input (resp. output) of instance `k` being the `k`-th "slice" of expression `Ak` (resp. of variable `Ak1`). The actual inputs and outputs (I/Os) are of greater dimensionality than the formal ones, because of the dimensions indexing the subsystem instance (which are the rightmost ones in the index lists [3]).

3: Static analysis of a system in isolation

Consider one of the systems of Fig. 1. It is of course not possible to ensure statically that it computes the expected result (to start with, the termination of such a computation is well known to be undecidable). However it is possible to define a few conditions which ensure that certain well-defined properties hold, and may be verified statically. The most SARE-specific of these conditions is the single assignment rule (SAR):

For each point of the domain of a variable, there is one and only one computation defining the value of the variable at this point.

In the following, we will say that a system or an equation is *valid* if it satisfies this rule. This section describes the static verification of the SAR within a single, parameterized system. We assume that this SARE is syntactically correct (all the variables are declared, all the dimensions match, etc).

3.1: Computation of the domain of an expression

An Alpha expression e denotes a data array, i.e. a collection of *values* defined over a *domain*. While the values usually depend on the inputs of the system, the domain, which we note $Dom(e)$, may be computed statically, by recursively applying the rules listed below. Notice that these rules are valid because they rely on operators preserving *DOM*.

Constants	$\boxed{Dom(c) = \mathbb{Z}^0}$
Variables	$\boxed{Dom(V) \text{ is declared in the header}}$
Unary operators	$\boxed{Dom(-e) = Dom(e)}$
Binary operators	$\boxed{Dom(e_1 + e_2) = Dom(e_1) \cap Dom(e_2)}$

The sum is defined where both expressions are defined.

[1] Instead of saying that a SARE "calls" another one, we prefer the term "use" which is free of the sequential connotation of "call".

Ternary operator

$$\boxed{Dom(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \bigcap_{i=1}^{3} Dom(e_i)}$$

The **if then else** is considered as a ternary operator and is defined where the condition and both alternatives are defined.

Restriction

D is a domain

$$\boxed{Dom(D : e) = D \cap Dom(e)}$$

This operator restricts an expression to the domain D.

Affine dependency

f is an affine function of the indices of the LHS

$$\boxed{Dom(e[f]) = f^{-1}(Dom(e))}$$

The value of $e[f]$ at point $\mathbf{z}$ is the value of e at point $f(\mathbf{z})$, hence the domain of $e[f]$.

case operator

$$\boxed{Dom(\textbf{case } e_1; ..; e_n; \textbf{esac}) = \bigcup_{i=1}^{n} Dom(e_i)}$$

This operator allows the piecewise definition of an expression by several subexpressions e_i with disjoint domains.

3.2: Equation analysis

Now consider an equation $V[i, j \ldots] = e$. To ensure that there is *at least* one computation defining the value of V for each point $(i, j, \ldots)$ of its domain, it is sufficient to ensure that:

$$Dom(V) \subset Dom(e)$$

where $Dom(V)$ is the domain of the variable V (in ALPHA, this domain is declared in the header of the system).

Practically, the analysis tool computes $D' = Dom(V) \setminus Dom(e)$, where $\setminus$ denotes the set difference. If D' is non empty, an error is issued, stating that V is not defined over D'. Thus not only does the tool detect the error, it also points it precisely to the user for correcting. Section 3.4 will show how this message may be made even more helpful.

Example: In the equation defining `Ar` in `ZeroColumn`, suppose line 5 was mistyped:

```
      {| i<K}     : A[i,j];
```

(notice `<K` instead of `<=K`). The analysis tool will output the following message:

```
ERROR: Variable Ar not defined over the domain: {i,j| i=K; 1<=j<=N}
```

3.3: Validity of the case statement

The **case** operator introduces the possibility of having more than one subexpression define the value of the same point of the domain. Therefore the analysis tool has to perform an additional check: if two of the **case** subdomains intersect, i.e.

$$D = Dom(e_i) \cap Dom(e_j) \neq \emptyset$$

then the **case** expression is overdefined on the domain D, which is reported as an error by the tool [2]. Since the **case** operator is the only possibility of introducing several definitions of the same value of a variable, the condition that subdomains have empty intersections is sufficient to ensure the *and only one* part of the single assignment rule.

[2] Actually there may be a surrounding restriction operator which restricts the **case** to a domain D' such that $D \cap D' = \emptyset$. In this case the expression is valid: to avoid flagging an error, the analysis tool has to maintain a *context domain* which is computed similarly as $Dom(e)$, but from the root of the expressions to their leaves. The error is issued only if the intersection of two **case** domains *and* the context domain is non-empty.

Example: in the same equation, if the second line of the **case** (line 6) was wrongly written:

```
      {| i>=K; j<=K}: 0[];
```

The analysis tool will output the following message:

```
ERROR: in case statement:  (...), domains of subexpressions overlap on:
{i,j | i=K; 1<=j<=K}
```

The previous analysis suffices to verify the single assignment rule, however there is other useful information which a static analysis may provide. For example it is useful to detect empty expressions[3], which are at best pointless (in a case statement) or a source of errors. To avoid cascaded error messages (if e_1 is empty, then from the rules above it is also the case of $e_1 + e_2$ and $D : e_1$), only the deepest empty subexpression needs reporting to the user.

Example: Still in the same equation, a mistake in line 5:

```
      {| i<=0}     : A[i,j];
```

will cause the following messages:

```
WARNING: This expression has an empty domain: {| i<=0} : A[i,j]
ERROR: Variable Ar not defined over the domain: {i,j| 1<=i<=K; 1<=j<=N}
```

3.4: Parameter related analysis

The previous examples illustrate how verbose and practically helpful these error detection techniques may be. In addition, it is even possible to refine the analysis by taking the parameters into account: suppose there was no restriction on the parameter `N` of the system `Gauss` (whose header would then be `system Gauss: {N |}...`). Obviously for negative values of `N`, all the variables of this system have an empty domain, which should be pointed to the user as a possible source of errors. The previous analysis will not detect such problems: extending the domains to $\mathtt{N} \leq 0$ does not change them, and therefore does not change the result of the analysis.

We now show how the analysis tool may prevent the user from writing a system which is not valid for some of the values in its parameter domain. This will prove useful for two reasons: it will help to detect more errors, and it will also allow the incremental validation of structured SAREs, as shown in Sect. 4.

All the previous domain checks come down to verifying the emptiness or non-emptiness of a domain D'. To ensure such a property for all the values of the parameters, we use the fact that the parameter domain is a canonical subspace of any domain appearing within a system: therefore we may compute the projection $proj_p(D')$ of D' on this parameter subspace. This is an orthogonal projection, therefore $proj_p(D')$ is a domain of DOM, which may then be compared to the declared parameter domain of the system (Fig. 2).

The following is the detail of the parameter related checks:

- The domain of an expression should be non-empty for all the values of the parameters. Let P be the parameter domain, $dim(P) = p$. Let $D = Dom(e)$ be the domain to verify, $dim(D) = n + p$. The domain $proj_p(D)$ is the set of the parameter values (permitted or not by the parameter space) for which the expression is non empty. The analysis tool therefore computes

$$D' = P \setminus proj_p(D)$$

[3]Here again the context domain of the subexpression has to be taken into account.

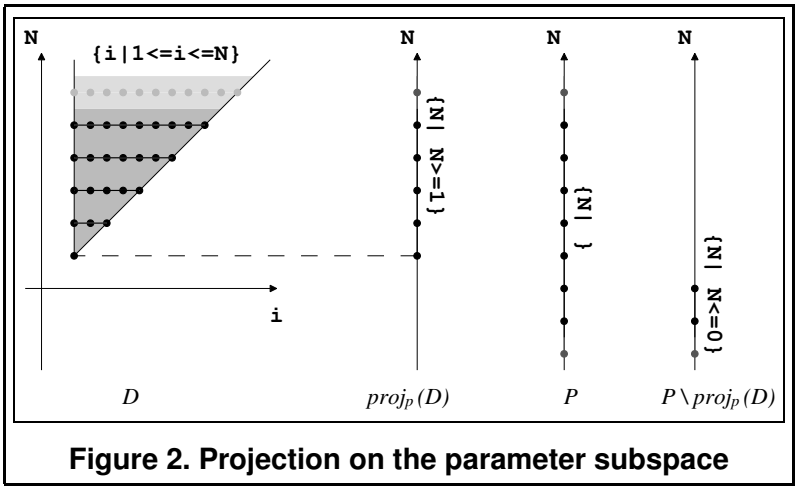

Figure 2. Projection on the parameter subspace

which is the set of values of the parameter spaces for which the expression has an empty domain. If $D' \neq \emptyset$, a warning is issued to state that the expression e is empty for the parameter values in D'.

Example: We may restrict the parameter `N` of the system `Gauss` to be positive:

```
12   system Gauss: {N | N>0}  ...
```

One may check that the system is valid even for `N=1`.

However the analysis will issue the following message:

```
WARNING: for parameters {N| N=1}, the expression Ak1 has an empty domain
```

- Similarly, the declared domains of output variables should be non-empty for all the values of the parameters (they do not necessarily appear on the RHS of an equation, unlike input and local variable which are tested for emptiness as they appear in an expression).
- The condition $Dom(V) \subset Dom(e)$ should be true for all the values of the parameters. Actually the proper check is already done in the previously described expression analysis, but we may make its output easier to read in case of a problem: after computing $D' = Dom(V) \setminus Dom(e)$ as previously, and if D' is non empty, we compute $proj_p(D')$, the set of parameter values on which there is a problem. If $P \setminus proj_p(D') = \emptyset$, there is a problem of definition for the whole parameter space. Otherwise, the problem might be parameter-related: the tool outputs an additional message stating that the problem only occurs for the parameter values in $D'' = P \cap proj_p(D')$, thus inviting the user to restrict the parameter space.

  Example: The following parameter domain for `ZeroColumn`:

  ```
  1   system ZeroColumn: {N,K| 0<=K<N}  ...
  ```

  will yield the following message (among others):

  ```
  ERROR: For parameters {N,K| N>0; K=0},
  variable Ar not defined over the domain: {i,j| 1<=i,j<=N}
  ```

- the intersection of two **case** sub-domains should be empty for all the values of the parameters. This problem is similar to the previous one and also allows a refinement of the error message.

Remark that a dual approach would be possible: using similar technique (projection, cut-set and union) we could compute the maximal domain of the parameters where the

system is valid, then impose it to the system. Our approach, however, is more helpful at the development stage. Besides it allows the programmer to be more restrictive than the maximal domain.

4: Static analysis of a collection of systems

4.1: Analysis of use statements

We consider now the case of a **use** statement, where a *caller* system uses a *subsystem*. The general form of such statement, with the notations of [3], is the following:

$$\textbf{use } D_{ext} \texttt{ subsystem}.f\ (e_1, .., e_m) \textbf{ returns } (v_1, ..v_n);$$

where D_{ext} is the (parametric) extension domain and f is the affine function assigning values to the parameters. We will note P_{sub} the parameter domain of the subsystem.

The analysis of this statement is deduced from its *substitution semantics* [3]: in short, a program containing a **use** statement is equivalent to one where this statement has been replaced with the body of the subsystem (properly modified to take into account the extra dimensions and the affine parameter assignation) and additional equations to perform the I/O passing: input equations relate the actual inputs and the formal ones:

```
SubSystemInputVariable = ActualInputExpression;
```

and output equations relate the actual outputs and the formal ones:

```
ActualOutputVariable = SubSystemOutputVariable;
```

Parameter checking Here again, we assume that the program is syntactically correct: the subsystem has been declared somewhere in the program, the correct number of actual inputs/outputs are given, their respective dimensions match the formal ones, etc.

The tool first checks, using the same techniques as before, that the extension domain is non-empty for all the values of the caller parameters.

Then it analyzes the values given to the parameters of the subsystem: they are defined by f which is an affine function of the caller parameters and, possibly, the extension indices (e.g. in the `Gauss` system, $f(\texttt{N}, \texttt{k}) = (\texttt{N}, \texttt{k})$ where `N` is a caller parameter and `k` is the extension index). The tool checks that, for all the possible values of the caller parameters, and for all the points in the extension domain, the values assigned to the subsystem parameters fall within the range permitted, i.e. within the parameter domain of the subsystem. This is expressed[4] by:

$$f(D_{ext}) \subset P_{sub}$$

As the *image* of an Alpha domain by an affine function is not always an Alpha domain (in the general case it is a *linearly bounded lattice*, as defined in [11]), we compute the convex hull of this image, which belongs to *DOM*: if $\mathit{ConvexHull}(f(D_{ext})) \subset P_{sub}$ then, since $f(D_{ext}) \subset \mathit{ConvexHull}(f(D_{ext}))$ we ensure $f(D_{ext}) \subset P_{sub}$. Otherwise an error message is issued. Since the points in $\mathit{ConvexHull}(f(D_{ext})) \setminus f(D_{ext})$ are only "holes" in the polyhedron, this algorithm only issues an error message when there is a problem on the polyhedron.

[4] In this expression, f as well as P_{sub} are parameterized by the callers' parameters.

As before, a verbose error message is obtained by computing the domain:

$$D' \;=\; ConvexHull(f(D_{ext})) \setminus P_{sub}$$

which is the set of disallowed values, and checking its emptiness.

Example: Line 17, the following **use** statement:

```
use {k| 0<=k<=N} ZeroColumn[N,k] (Ak) returns (Ak1);
```

will cause the following message:

```
ERROR : in statement ‘‘use ... ZeroColumn ... ’’,
parameter values in {N,K| K=0, N>=1} not allowed
```

Obviously, the more restricted the parameter domain of the subsystem, the more acute the checks performed here.

Input/Output checks The substitution semantics implies that the verifications to be performed on the I/Os of a subsystem **use** may be deduced from those of the equations described in 3.2. There is slightly more work involved, however, since the domains of the added variables are transformed as described in [3], but no new technique is involved, and we will not describe the details here.

Validity of the use statement Suppose that the three previous conditions are satisfied:

1. the subsystem is valid on its parameter domain.
2. $f(D_{ext}) \subset P_{sub}$.
3. the virtual I/O passing equations are valid.

It is then possible to *inline* this **use** statement thanks to its substitution semantics [3]: the **use** is replaced with several equations, of two kinds: some are the I/O passing equations, and are valid (third condition). The other ones are the body of the subsystem, transformed according to the substitution semantics. Since all the equation transformations involved in this process preserve the single assignment rule, these equations are valid, too (first condition). By definition of the **use** semantics, both programs (the one containing the **use** and the inlined one) are equivalent. Since they only differ by the previous equations, we conclude that the three conditions above define the validity of the **use** statement.

4.2: Global analysis of a collection of systems

Now we may describe a general down-top validation method for a complete program. Such a program is an acyclic graph of ALPHA systems using each others. Systems without a **use** statement among their equations are called *sinks*. A system using a subsystem is called a *predecessor* of this subsystem.

- First, the sinks are analyzed, and their parameter domain is restricted as much as possible. No warning message should remain. For example, for the system `ZeroColumn`, we have to restrict `K` and `N` at least to the domain given Fig. 1. It is possible, however, to constraint the parameters more than what the analysis suggests, for example we could put a higher bound on `N` depending on the intended application.
- Then the predecessors of the leaves are analyzed. If they are written to use a sink with invalid values of its parameters, the tool will spot it and the programmer will be encouraged either to correct the error, or to restrict more the caller parameters. Meanwhile, the other equations of the caller are also analyzed, with the same effect.

- This process is repeated on the parents of the parents, and so on until the whole program passes the static analysis.

5: Conclusion

We have presented an automatic tool for the analysis of structured systems of affine recurrence equations. This tool has been implemented in the ALPHA environment. For each system of the program, it checks that the single assignment rule is satisfied, helps define strong correctness conditions on the parameters, and prevents this system to be used with incorrect values of these parameters, thus allowing the incremental specification and validation of a hierarchy of SARE.

All the errors won't be detected: typically the program consisting of a single equation $A = A$ is considered valid by our analysis tool, for it satisfies the single assignment rule. Whether a SARE actually does something is the SARE version of the termination problem, and is of course undecidable. However we found that usually a program works as soon as it passes the static analysis.

These methods are obviously portable to any system based on the formalism of SARE. Similar static analysis techniques should also apply to more general formalisms, such as that of linearly bounded lattices as defined in [11], where in fact some tests can be made more precise.

References

[1] M. Berry and A. Sameh. Multiprocessor jacobi algorithms for dense symmetric eigenvalue and singular value decompositions. Technical report, Center for supercomputing research and development, 1986.

[2] R. P. Brent, F. T. Luk, and C. Van Loan. Computation of the singular value decomposition using mesh-connected processors. *J. VLSI Comput. Syst*, 1:250–260, 1985.

[3] F. de Dinechin, P. Quinton, and T. Risset. Structuration of the alpha language. In *Massively Parallel Programming Models*, Berlin, Germany, August 1995. IEEE.

[4] Jack B. Dennis. Automatic mapping of stream-processing functional programs. In *Massively Parallel Programming Models*, Berlin, Germany, August 1995. IEEE.

[5] C. Dezan, E. Gautrin, H. Le Verge, P. Quinton, and Y. Saouter. Synthesis of systolic arrays by equation transformations. In *ASAP'91*, Barcelona, Spain, September 1991. IEEE.

[6] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[7] F. T. Luk. A triangular processor array for computing singular values. *Lin. Alg. Appl.*, 77:259–273, 1986.

[8] Christophe Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[9] P. Quinton, S. Rajopadhye, and D. Wilde. Derivation of data parallel code from a functional program. In *IPPS*, Santa Barbara, USA, April 1995.

[10] S. Rajopadhye and D. Wilde. The naive execution of affine recurrence equations. In *ASAP*, Strasbourg, France, July 1995.

[11] J. Teich and L. Thiele. Partitioning of processor arrays : a piecewise regular approach. *Integration, the VLSI journal*, 14:297–331, 1993.

[12] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[13] D. Wilde. A library for doing polyhedral operations. Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993. Also published as INRIA Research Report 2157.