Structured Systems of Affine Recurrence Equations and their Applications

FLORENT DE DINECHIN
IRISA

Methodologies based on *systems of affine recurrence equations* (SAREs) may be used in the fields of computer-aided parallel programming and hardware synthesis, allowing automatic program *analysis* and *refinement*. This paper addresses a current limitation of these methodologies: the lack of the SARE equivalent to program structures. We therefore introduce structured systems of affine recurrence equations in the framework of the ALPHA language. We show how a a complex SARE may be structured into several SAREs of simpler dimensionality, while retaining the possibility of analyzing and automatically transforming such a program using polyhedral computation. This leads to new methodologies for both hardware synthesis and parallel programming, some of which are demonstrated in this paper.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Automatic Synthesis and Hardware Description Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*; J.6 [**Computer-aided engineering**]: Computer-aided design

General Terms: Design, Languages

Additional Key Words and Phrases: affine recurrence equations, polyhedral computation, program and data structures, regular processor arrays, static analysis

Introduction

Context. A computation, in the most general sense, may be viewed as a set of basic operations relating a set of data. In order to express a computation in a form which is more compact than the extensive enumeration of these operations, these sets are usually given a *structure:* vectors, arrays, lists and trees are example of data structures while `do...loop` constructs are examples of computation structures. Modern programming languages express this need for structures.

Implementing an computation may be a different issue from expressing it: the formula $c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$ is a high-level expression of the matrix product, with a range of possible implementations. In particular, when looking for a parallel implementation, there are usually several ways to schedule the basic operations (here additions and multiplications), some of which being simply impractical, and some other yielding a variety of tradeoffs between software and hardware. The exploration of this implementation space is a very complex issue. The language and tools described in this paper address this issue.

We see at least two requirements for this exploration to be possible. The first is that the computation or algorithm is expressed at the highest possible level of abstraction, which means in particular that it is disconnected from any specific implementation. The previous sum for the matrix product is valid whatever the number or type of processors or multipliers we will use to implement it, and in

this respect is a high-level description. The second requirement is that we need to be able to *analyze* algorithms in an automatic (or at least assisted) way, in order to automate (or at least guide) the *synthesis* of their implementation. Such an analysis is only computationally possible on strongly structured algorithms, and here comes another tradeoff: the more structured the data and the computations, the easier their analysis but the more restricted the set of expressible algorithms. One of the reasons for the inefficiency of current "parallelizing compilers" is the large expressive power of their input (general-purpose sequential languages such as C or Fortran) which makes it difficult to analyze programs accurately.

In this paper we chose the complimentary approach: we restrict the language – and thus the class of algorithms which can be efficiently expressed – to favor program analysis and manipulation. More precisely, the data structures are restricted to have an Euclidean topology; this includes vectors and matrices, for example, but no trees nor graphs. The dependencies between computations are also restricted to have a certain regularity in this topology. These are significant restrictions, but any of the remaining algorithm may then be analyzed and transformed automatically in an efficient interactive parallelization process. This restricted class of algorithm still contains a significant part of the computation-intensive application cores needed in fields such as signal and image processing and scientific computation.

Affine Recurrence Equations. This philosophy is the basis of the formalism of *affine recurrence equations* (AREs) first introduced by Karp, Miller and Winograd [1967]. AREs relate data arrays of arbitrary dimension, using classical computational operator to express and transform the values held by these arrays, but also *spatial* operators to express and transform their shapes. An algorithm is then expressed as a system of such equations, or SARE. There have been several refinements of this model, mostly defining the possible shape of the data arrays and the related restrictions on the spatial operators. This paper deals with the *polyhedral* SARE model, where the data arrays are defined over unions of convex polyhedra, while the spatial operators are basically affine functions in the index space. Since a convex polyhedron can be expressed as the intersection of several *affine* half-spaces, both polyhedral domains and affine spatial operators will be subject to analysis and transformations using classical linear algebra and linear programming tools.

In this paper, we will express SAREs in the ALPHA language which was designed as a convenient syntax for the polyhedral SARE formalism. An environment, MMALPHA, built on top of Wolfram's Mathematica, provides automatic tools for the analysis and transformation of programs. This environment may be used for example to engineer hardware and software as follows: an algorithm is first expressed at a very high level as a SARE (for example as the previous sum $c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$). Then this SARE is analyzed and transformed in order to synthesize a lower-level – but semantically equivalent – implementation which matches the requirements of the application. A complete example of such a design process will be given in section 3.5. Automatic or semi-automatic program transformations may lead to VLSI regular arrays [Dezan et al. 1991], imperative code [Rajopadhye and Wilde 1995], or a collection of loop nests aimed at a massively parallel architecture [Quinton et al. 1995]. The theoretical and technical bases for such a *refinement* process are the result of researches spread over thirty years [Karp et al. 1967; Kung and

Leiserson 1978; Kung 1982; Moldovan 1982; Quinton 1984; Rao 1985; Delosme and Ipsen 1986; Rajopadhye and Fujimoto 1987; Yang and Choo 1991; Teich and Thiele 1993], demonstrated and validated thanks to simple core examples (simple matrix operations, filters, convolutions...).

Motivations and goals. Automated tools like MMALPHA now make it possible to handle algorithms of a complexity much greater than those core examples studied by paper and pencil in the literature. However, in order to deal with real-world problem, the SARE formalism lacks structuring techniques which would allow to break up a complex specification into a hierarchy of simpler entities: so far any program or algorithm, whatever its complexity, has to be expressed as one single system of equations. The number of equations, but also the high dimensionality of the data arrays may make the writing of such a system a very difficult task. For example, the singular value decomposition (SVD), a matrix computation commonly used in signal processing applications, involves data arrays of dimension 5 (this number is the depth of the loop nests in a sequential implementation).

The purpose of this paper is therefore to extend the formalism to handle more complex SAREs, thanks to hierarchical structuring techniques. The main motivation is to benefit from the well-known advantages of structured programming. Some other benefits, however, are more specific to the SARE world:

—The analysis of a SARE, when carried on separate, smaller units, may be both simpler and more powerful.

—When translating a SARE into some target language for implementation, it is possible to exploit the structures of this language and match them to the structures of the SARE.

—It is possible to define libraries of SAREs with all the advantages of libraries of code or libraries of hardware component, and more: such libraries are much more flexible because their components may be refined *in context*, as this paper will show.

Why not use classical program structuring techniques? There is a wide range of structure constructs available in the two major computational models: sequential programs may be structured as procedures and functions, while functional programs are intrinsically structured as functions, which may be used through well studied higher-order operators such as *map*. In addition, in the field of VLSI synthesis, there are hardware description languages which are structured in terms of components. None of these approaches, however, suits well the SARE formalism: we need program structures to be constrained within the affine/polyhedral world in order to retain the strong point of the formalism, the ability to automatically analyze and transform programs.

This paper thus introduces a program structuring technique specifically adapted for SAREs: we show how a computation may be described as a structure of systems of affine recurrence equations (SSARE), in such a way that the structure is very high-level (implementation-independent) and subject to formal manipulation by the existing analysis and refinement tools. Besides, the structures we introduce will be interpretable as:

—function or procedure calls within affine loop nests, if the SSARE is interpreted as sequential code,

—regular component instantiation if it is interpreted as parallel hardware,

—constructs similar to, but more general than those of the languages based upon map-like parallelism, if it is interpreted as a data-parallel program.

Outline. This paper is organized as follows: the first section introduces the classical SARE formalism in the polyhedral model, using the ALPHA language as a syntax. It emphasizes the strong points of the model, the ability to analyze and transform programs, and how it allows to transform formally a high-level specification into an implementation. The second section presents the extensions to this model, defining structured systems of affine recurrence equations within the polyhedral model. The last section discusses the use of SARE structures in program refinement and gives several examples of their applications in the current MMALPHA tool.

1. SIMPLE SARES IN ALPHA

This section gives a definition of the syntax and semantics of the ALPHA language. ALPHA being basically a convenient notation for the formalism of polyhedral affine recurrence equations, we first define polyhedra and space variables, the basics of this formalism. Then we introduce in 1.2 the ALPHA expressions and give their compositional semantics. Subsection 1.3 defines the syntax and semantics of an affine recurrence equation, and subsection 1.4 defines a system of such AREs.

The first parts of this section are illustrated by the very classical example of an ALPHA matrix-vector product given below as Prog. 1.

```
1   system matvect (M : {i,j | 1<=i<=10; 1<=j<=10} of real;
2                   V : {j | 1<=j<=10} of real)
3         returns (R : {i | 1<=i<=10} of real);
4   var
5     C : {i,j | 1<=i<=10; 0<=j<=10} of real;
6   let
7     C = case
8           {i,j | j=0} : 0.(i,j->);
9           {i,j | j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
10        esac;
11    R = C.(i->i,10);
12  tel;
```

Program **1:** Matrix-vector product

In the description of the syntax of the language, we use the following conventions :

*phrase**	denotes zero or more repetitions of *phrase*.
phrase1 | *phrase2*	denotes alternation, either *phrase1* or *phrase2*.
[. . .]	denotes optional phrase.
(. . .)	denotes syntactic grouping.
`courier`	denotes a terminal.

Italic denotes a non-terminal.

1.1 Basics: array domains and variables

The main data structure in ALPHA is the *polyhedral data array*. Such an array is a function of some integer vector space $\mathbb{Z}^n$ to a scalar value space (integer, boolean or real). The domain of this function is a convex polyhedron of $\mathbb{Z}^n$, which has in ALPHA the following syntax:

Domain :: { *IndexList* | *ConstraintList* }

Intuitively, this data structure enables to represent vectors, matrices, any n-dimensional array, but also more complex data shapes such as triangular matrices (e.g. {`i,j| 1<=i,j<=10; j<=i`}). Infinite domains are also allowed to represent, for example, infinite data-streams (e.g. {`t| t>=0`}).

ALPHA variables hold such polyhedral data arrays: thus a variable v is declared by specifying its domain and the type of its values. For example, in Prog. 1, line 5 declares a variable named `C` which is a square 10×10 matrix.

Actually the domains in ALPHA may be the union of finitely many convex polyhedra. The set *DOM* of all the finite unions of integral convex polyhedra has the following fundamental property [Wilde 1993]:

PROPERTY 1. *DOM is closed under intersection, union, set difference, preimage by an affine function, and convex hull of the image by an affine function.*

This property is at the core of program analysis and refinement, as the rest of this section will show. Besides the domain operations listed here have been implemented in Wilde's polyhedral library [1993] which is the main engine of the MMALPHA environment.

ALPHA is also strongly typed with respect to the scalar type of the values held in polyhedral data arrays (integer, real or boolean), but we will not detail this scalar typing, which is based on very basic and classical type inference techniques.

1.2 Expressions

ALPHA expressions are built out of the previous variables, and constants, the latter being also considered as polyhedral data arrays of dimension zero. An expression also holds a polyhedral data array, which is defined compositionally from those of its subexpressions thanks to property 1.

There are two kinds of operators: *pointwise* computation operators transform the values, and *spatial* operators manipulate the domains of the data arrays. In addition, the *reduction* operator is both computational and spatial.

1.2.1 *Pointwise operators.* A pointwise operator describes a computation applied to all the points of a data array. For example, if E_1 and E_2 are two expressions of the same dimension, then $E_1 + E_2$ is an expression of the same dimension, defined anywhere where both E_1 and E_2 are defined, and whose value at each point is the sum of the values of E_1 and E_2 at this point.

More formally, the domain of $E_1 + E_2$ is the intersection of the domains of E_1 and E_2, and its value function is the sum of those of E_1 and E_2.

Other pointwise operators include most unary and binary arithmetic and logical operators, comparison operators, plus the ternary `if then else` operator.

1.2.2 *Spatial operators.* The *restriction* of the expression E to the domain D is noted $D : E$. It is an expression whose values are those of E, but whose domain is the intersection of the domain of E with D.

The *case* operator allows the piecewise definition of an expression by several subexpressions defined over disjoint domains. For example `case`E_1; E_2; `esac` is an expression whose domain is the union of the domains of E_1 and E_2. Its values are defined as those of E_1 over the domain of E_1, and those of E_2 over the domain of E_2. If both values are defined on some point, this expression is undefined. To avoid this situation, the restriction operator is usually needed to restrict E_1 and E_2 properly. See Prog. 1 for an example.

Finally, the *dependence* operator establishes an affine mapping from the points of a domain to the points of another domain. Typical affine dependencies are translations or value broadcast. If f is an affine function, $E.f$ is an expression whose value at a point $\mathbf{z}$ is the value of E at the point $f(\mathbf{z})$, and whose domain is therefore the preimage by f of the domain of E.

The ALPHA syntax of affine functions is:

AffineFn :: `(` *IndexList* `->` *IndexExpList* `)`

where the index expressions relate the indices in the index list.

Notice that a constant is a data array of dimension zero, and therefore usually needs a dependence function to be turned to a data array of greater dimension. For example, if `A` is a matrix and we want to add 42 to each element of `A`, then `A + 42` is incorrect, for it is the pointwise sum of an object of dimension 2 and an object of dimension 0. The correct way to express this is to build a two dimensional array of 42s using a dependency operator, as in `A + 42.(i,j->)`. See also line 8 of Prog. 1.

1.2.3 *The reduction operator.* This operator allows to write high-level expression such as the summation of the introduction. Let us take another example, our matrix-vector product of Prog. 1. It could be expressed in a more abstract manner as $R_i = \sum_{j=1}^{N} M_{ij} V_j$. This sum is expressed in ALPHA as the following line, which could replace lines 7 to 11:

```
R = reduce( +, (i,j->i), M*V.(i,j->j) );
```

The reduce operator needs a binary associative and commutative operator (here `+`), an ALPHA expression (here `M*V.(i,j->j)`) of domain D (here a matrix) and an affine projection f (here `(i,j->i)`) specifying in which direction(s) of D the reduction is to take place. The domain of this `reduce` expression is the image of D by this projection, and the value of each point of this image is the combination of all its antecedents by f in D.

Reduction operators allow high-level algorithmic expression. An automatic program transformation, called *serialization*, expands the line above into lines 7-11 of Prog. 1. This transformation is an example of program refinement: it specifies an order of the computations (here along increasing `j`) which is an implementation choice (serializing along decreasing `j` would also be possible).

Finally the general syntax of an expression is:

Expression ::

Variable | *Constant*	(Terminals)
| *UnOp Expression*	(Pointwise unary)
| *Expression BinOp Expression*	(Pointwise binary)
| `if` *Expression* `then` *Expression* `else` *Expression*	(Pointwise `if`)
| *Domain* `:` *Expression*	(Restriction)
| `case` *ExpressionList* `esac`	(Disjunction)
| *Expression* `.`*AffineFn*	(Affine dependency)
| `reduce(`*BinOp* `,` *Expression* `,` *AffineFn* `)`	(Reduction)
| `(` *Expression* `)`	

1.2.4 *Static analysis of expressions.* The previous definitions have shown informally that, using property 1, it is possible to compute, *for any* ALPHA *expression,* the polyhedral domain where it is defined (for a more formal approach see [de Dinechin and Robert 1996]). This key feature of the language is implemented in the MMALPHA environment, and is the groundstone of most tools for program analysis and refinement. We illustrate it here by demonstrating one of these tools for the detection of bounds errors. Another example will be given in section 3.5.1.

In an array language such as ALPHA, most errors which are not purely syntactic are bound errors. For example, the branches of a `case` statement should have disjoint domains, for if they do not, then there are conflicting definitions of the value of the `case` expression for the corresponding points. Here is a trivial example of this situation:

```
A = case
     {i,j,k| i<=k}      : 0.(i,j,k->);
     {i,j,k| i>k; j<=k} : 1.(i,j,k->);  -- defined for j=k
     {i,j,k| i>k} j>=k} : 2.(i,j,k->);  -- also defined for j=k
    esac;
```

What should be the value for `j=k`, `1` or `2`? Obviously this equation, although syntactically correct, is ill-defined.

Now thanks to property 1, it is always possible, for such a `case` expression, to compute the pairwise intersections of the domains of the branches. It is also possible to check that this intersection is empty. If it is not, as in our example, then there is a definition conflict. The domain of this confict is precisely this intersection, which allows for very accurate error messages.

In our example, invoking the MMALPHA `analyze[]` tool yields the following message:

```
ERROR: in case statement, domains of subexpressions overlap on
{i,j,k | j=k}
```

This trivial example shows that, thanks to accurate polyhedron computation, the error message is helpful enough to spot the problem easily. Several other similar checks have been implemented in the extensive `analyze[]` tool [de Dinechin and Robert 1996].

1.3 Equations

An ALPHA program is basically a system of equations, each equation having a variable as left-hand side and an expression as right-hand side.

The usual syntax of an equation is therefore:

Equation :: *Identifier* = *Expression* ;

It is important to distinguish such an equation from an affectation: the ALPHA equation expresses an identity between both sides. In particular, to be valid, an equation must define the whole of the variable, that is, the domain of the expression must at least cover the domain declared for the variable.

1.3.1 *Array notation.* In most cases, the dependence operator may be rendered with an easier-to-read array notation [Wilde 1994]. This notation is nothing but a simpler syntax for an equation which has several index declarations within domains affine functions on its right-hand side (e.g. `i,j` in lines 8-9 of Prog. 1). If the dimensions are coherent, then these declarations are similar (the names of the indices may differ, but their number is the same), and the equation may be simplified by putting them on the left-hand side. An example of this is Prog. 2, which is the core of Prog. 1 written in array notation.

```
7    C[i,j] = case
8                {| j=0} : 0[];
9                {| j>0} : C[i,j-1] + M[i,j] * V[j];
10               esac;
11    R[i] = C[i,10];
```

Program **2:** Array notation for the matrix-vector product

This array syntax is no longer compositional, and thus may not be used in all cases, but it will be adequate for the rest of this paper. The reader, however, should keep in mind that square bracket are nothing but a convenient notation for affine dependency functions.

1.3.2 *Equation analysis.* There are several possible analyses which can be performed on equations. For example, it is possible to compare the domain of the variable and the expression to check whether the variable is defined on the whole of its declared domain. For this purpose, we compute the set difference of the domain of the expression minus the domain of the variable (the set of finite unions of convex polyhedra is closed under domain difference). If this difference is not empty, then an error is issued, pointing this very domain as being the set of points lacking a definition [de Dinechin and Robert 1996].

The converse is also possible: using similar techniques, we may check whether the domain of the expression is bigger than that of the variable. If it is, then the equation describes useless computations. Fore some interpretations (or implementations) of ALPHA programs, this is the basis for automatic optimization.

These examples show again that exact domain computation is central to the language and should be preserved by the program structures we will introduce in section 2.

1.4 Systems

An ALPHA system is a set of mutually recursive equations. All the variables are declared in the header. There are three classes of variables. Input variables only appear on the RHS of the equations: their domain is declared in the header of the system, but they do not have an equation defining them. Output variables are returned by the system. Local variable are auxiliary variables. The syntax of a system is thus:

SystemDecl ::
  `system` *Name* `(` *InputDeclList* `)`
    `returns` `(` *OutputDeclList* `)` `;`
    [`var` *LocalDeclList* `;` *]*
    `let`
    EquationList
    `tel`

Each declaration list contains declarations of variables as described in 1.1.

1.5 A simple example of program refinement

This section does not intend to give an extensive overview of manipulating SAREs, but rather to show how the affine/polyhedral model makes it possible to automate program transformation (the previous section showed how it yielded powerful and accurate analysis techniques). Our example of program refinement leads from an abstract specification to a processor array suitable for VLSI synthesis. The main purpose, however, is to justify the rather exotic definition, in the next section, of program structuring constructs based purely on polyhedra and affine function to retain this possibility.

A program like that of Prog. 1 is first *uniformized* [Rajopadhye et al. 1986; Yaacoby and Cappello 1988] to remove the data broadcasts and non-local communications. Such non-uniform data dependencies are simply detected as non-translation affine functions, which may be replaced with a pipeline of values. Reduction operators are serialized as well. In both cases, there may be several choices for the directions of the pipelines introduced, which are currently left to the user.

When the program is uniform, all the computations live in the same index space and only depend on neighboring computations. It is then possible to compute a *schedule* of these operations: each point of this index space is assigned an affine execution date consistent with the data dependencies. For this purpose the dependencies are automatically extracted from the ALPHA program and expressed as affine constraints, the polyhedral computation domains give other affine constraints, yielding a linear integer programming optimization problem solvable by computer [Feautrier 1992; Darte and Robert 1994].

Then an affine *change of basis* is performed on the index space of each variable: their domains are transformed (still using property 1) so that one of the indices becomes the *time* at which this variable is computed, and the other indices specify the *processor* on which the computation is performed, in some processor array whose shape is given by the resulting domains of the variables.

When this process is carried on Prog. 1, we get Prog. 3, which is an abstract description of the architecture pictured by Fig. 1.

```
1        system matvect (M : {i,j | 1<=i,j<=10} of real;
2                        V : {j | 1<=j<=10} of real)
3              returns (R : {i | 1<=i<=10} of real);
4        var
5          ML : {t,p | p+1<=t<=p+10; 1<=p<=10} of real;
6          VS : {t,p | p+1<=t<=p+10; 1<=p<=10} of real;
7          C : {t,p | p<=t<=p+10; 1<=p<=10} of real;
8        let
9          ML[t,p] = M[p,t-p];
10         VS[t,p] =
11             case
12               {| p=1} : V[t-1];
13               {| p>1} : VS[t-1,p-1];
14             esac;
15         C[t,p] =
16             case
17               {| t=p} : 0[];
18               {| t>p} : C[t-1,p] + ML[t,p] * VS[t,p];
19             esac;
20         R[i] = C[i+10,i];
21       tel;
```

Program **3:** ALPHA program interpreted as a systolic circuit

In this program, the data arrays are still two-dimensional as in Prog. 1, but now the index `t` represents the time and the index `m` is the processor index in the linear array of Fig. 3. The declaration domains of the variables show that there are `10` processors (indexed by `p` such that `1<=p<=10`), and that the `m`-th processor computes for `p<=t<=p+10`. Line 12 shows how `V` is input at time `t` on the first processor (`p=1`), and line 13 shows how it is then propagated from processor `p-1` to processor `p` through one register (implied by the `t-1` dependency). In the figure, the registers are drawn as boxes. The computation equation of line 18 is interpreted as hardware operators, whereas the data translation equations are interpreted as registers. Finally the last equation shows that the `i`-th bit of the result is output by the `i`-th processor at time `i+10`. The complete program thus describes a virtual linear array (Fig. 1).

This design is still very abstract. Additional lower-level program transformations are needed to turn control information present in the domains into systolic control variables [Teich and Thiele 1991]. The resulting ALPHA program may then be translated [Le Moënner et al. 1996] into structural VHDL for synthesis by commercial VLSI CAD tools like Compass or Synopsys. We will come back on this translation later in 3.3.

2. STRUCTURED SARES

This section extends the SARE formalism and the ALPHA language to allow program structuring. For this we first address in 2.1 the issue of *genericity* of a system. Then in 2.2 we introduce SARE structures, their syntax in ALPHA and their se-

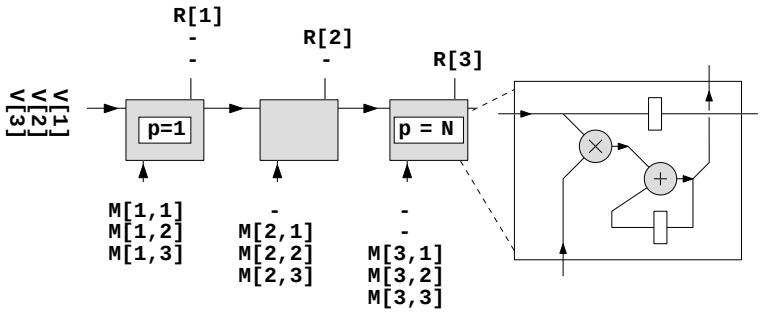


Fig. 1. Systolic array for the matrix-vector product

mantics.

2.1 Size parameters

Section 1 showed a matrix-vector product of fixed size 10. This obviously prevents a system like Prog. 1 to be used in a more complex application in an useful way, for an other application will need matrix-vector products of different size, the size even varying within the application. What is needed is therefore a more *generic* system, describing a matrix-vector product of arbitrary size, say N. The actual size should be an implementation detail.

The polyhedral SARE model allows for a simple but powerful parameter scheme: all we need is add an index `N` to each polyhedron of the system, and consider this index as a size parameter. For example the domain {`i,j | 1<=i,j<=10`} becomes {`i,j,N | 1<=i,j<=N`}. This index also needs adding in affine functions to replace the 10s there. Thus the dependency `(i->i,10)` becomes `(i,N->i,N,N)`. The `N` on the right-hand side is needed for dimension consistency. It is easy to show that the modified system is a valid ALPHA program, and that it can be analyzed and transformed as well (we will come back on this later).

Now we have added a "`,N`" to all the index lists of polyhedra, and to the left-hand and right-hand sides of all the affine functions. There is a lot of redundancy there, because this parameter index actually stands for a *global* constant. Therefore it makes sense to declare it only once, in the beginning of the system, and then omit its declaration in the rest of the system. Besides, when declaring it, we may as well declare a domain of values permitted for this parameter, as a usual polyhedral domain.

This defines the syntax for parameterized ALPHA systems, exemplified by Prog. 4. More generally, the parameter domain may be any ALPHA domain, in particular there is no limit on the number of parameters, and it is possible to express any affine constraint between the parameters. For example, {`M,N | M<2N`} is a valid parameter domain. Parameters may then appear anywhere in the system where indices are allowed, that is, in domains (see lines 2-4 of Prog. 1) as well as in affine functions (see line 12). This parameterized syntax obviously retains all the properties of the language, since it is always possible to write the parameterized system as a non-parameterized one.

```
1   system matvect {N| N>1}
2                  (M : {i,j | 1<=i,j<=N} of real;
3                   V : {j | 1<=j<=N} of real)
4          returns (R : {i | 1<=i<=N} of real);
5   var
6     C : {i,j | 1<=i,j<=N} of real;
7   let
8     C = case
9           {i,j | j=0} : 0.(i,j->);
10          {i,j | j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
11        esac;
12    R = C.(i->i,N);
13  tel;
```

Program **4:** Parameterized matrix-vector product

2.1.1 *Parameter related analysis.* With respect to program analysis and transformation, however, parameterization is more than simply syntactic sugar: parameters are syntactically mere indices, but their semantic is that of a constant in a system. For example, the parameterized affine function `(i,j->i+N, j+N)` is a translation and should be considered as such in the uniformization process, although its closed affine form `(i,j,N->i+N, j+N, N)` is not a translation.

Parameters also increase the efficiency of the previous static analysis tools: although the main techniques of domain inference are the same, it is possible for example to project the domain where an error occurs on the parameter subspace (such a projection also yields a domain) and compare this domain with the declared parameter range. Thus it is possible to verify that a property holds on the whole of the parameter range, or to compute precisely the set of parameters where it holds [de Dinechin and Robert 1996].

Optimization of the actual value of a parameter, in a given program refinement context, will also be possible in this framework, although this possibility has not been studied in depth yet.

2.2 Structures

The issue of structuring a complex algorithm into a hierarchy of SAREs is more complex than it seems. Obviously, it is partly addressed by the decomposition of the problem into equations: it is always possible to break an equation into two simpler equations, using an auxiliary variable to hold a subexpression of the initial expression. The reverse operation, replacing a variable appearing in an expression with its definition, is also always possible. The MmAlpha environment provides assistance for both these operations, thus ensuring that the resulting system is equivalent to the initial one (these operations are similar to the β-conversion in the lambda-calculus).

2.2.1 *The need for higher-order program structures.* Structuring using equations, however, basically remains first-order structuring (in the usual functional meaning), and is thus limited. These limits appear more clearly on the following example: suppose we want to write in Alpha an algorithm for a signal-processing

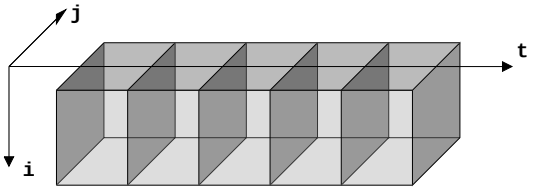


Fig. 2. Linear collection of bidimensional matrices

application, which involves time-varying matrices. Such matrices are represented in ALPHA as three-dimensional data arrays (two matrix dimensions, and one – possibly unbounded – time dimension) as represented on Fig. 2.

Now let us try to re-use the equations of the matrix-vector product to operate on these time-varying matrices. Obviously it is not possible in a straightforward manner, for the inputs don't have the proper dimension. For example we will need to rewrite the whole of the equation defining `C` to add one dimension to the domains and the affine functions, as shown by Prog. 5.

```
8    C = case
9          {i,j,t | j=0}: 0.(i,j,t->);
10         {i,j,t | j>0}: C.(i,j,t->i,j-1,t) + M * V.(i,j,t->j,t);
11       esac;
```

Program **5:** Using the matrix-vector product on time-varying matrices

Structuring using variable will never be adequate when such a *dimension extension* is needed, which is a very common case: it corresponds to a procedure or function call within a loop nest in imperative languages, or to a `map` structure in functional languages.

2.2.2 *Introductory example.* As another, more concrete example, let us try and write ALPHA SAREs for the addition and multiplication of two integers (or fixed-point numbers) written in binary notation. These numbers will be coded as arrays of booleans in ALPHA.

The base block of these operation is the *full adder* function which takes three binary inputs A, B and C_{in} and expresses their sum on two bits X and C_{out}:

$$A + B + C_{in} \;=\; X + 2C_{out}$$

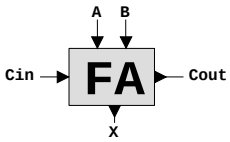
Prog. 6 is a full adder function written in ALPHA. In this system, the domains of all the variables are $\mathbb{Z}^0$ (one point).

An adder is classically described as a sequence of full adders with carry propagation, (hence the names C_{in} and C_{out}, for "carry") as shown on Fig. 3.

In order to re-use the system given as Prog. 6, we need to keep the same equations, but to change the domains of the variables so that they become arrays of bits:

```
A,B,Cin,Cout,X : {b| 0<=b<W} of boolean;
```

```
system FullAdder (A,B,Cin : boolean)
         returns (X,Cout : boolean);
let
  X= A xor B xor Cin;
  Cout= (A and B) or (A and Cin) or (B and Cin);
tel;
```



Program **6:** Full Adder system

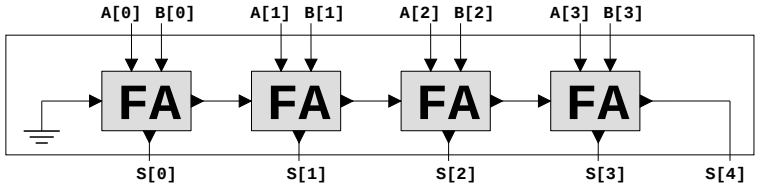


Fig. 3. An adder

The structure construct `use` in ALPHA allows precisely that: it allows a system to be used by another one with an *extension of the dimensionality* of the subsystem. This dimension extension is expressed as an ALPHA domain, in our example `{b| 0<=b<W}`. An example of this feature is Prog. 7.

```
1       system Plus: {W|W>1} (A,B: {b| 0<=b<W} of boolean)
2                    returns (S : {b| 0<=b<=W} of boolean);
3       var
4         Cin, Cout, X : {b| 0<=b<W} of boolean;
5       let
6         Cin[b] =
7           case
8             {| b=0} : 0[];
9             {| b>0} : Cout[b-1];
10          esac;
11        use {b| 0<=b<W} FullAdder[] (A,B,Cin) returns(X, Cout);
12        S[b] =
13          case
14            {| b<W} : X;
15            {| b=W} : Cout[W-1];
16          esac;
17      tel;
```

Program **7:** Addition using subsystem `FullAdder`

In this system, the line 11 reads as follows: "Use a *collection* of instances of the subsystem `FullAdder`. This collection has the shape of the extension domain `{b| 0<=b<W}` and is thus indexed by index `b`. Let the inputs of the `b`-th instance be the variables `A`, `B` and `Cin` at point `b`, and similarly let the outputs of this collection of instances be the variables `X` and `Cout`."

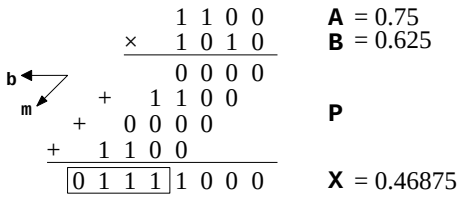


Fig. 4. Product of two fixed point reals in binary representation

The lines 6-10 describe the carry propagation, and lines 12-16 define the output of this binary adder.

2.2.3 *Inputs and outputs.* In Prog. 7, the actual inputs given to the system (`A`, `B` and `Cin`) are all variables. In general, however, they may be any expression: their semantic is that of a right-hand side of expression, and the link between actual and formal inputs is a virtual *equation* (in the usual ARE sense) [de Dinechin et al. 1995].

Actual outputs, on the other hand, have a left-hand side semantics, and so they have to be variables only.

Here is the general syntax of a `use` construct. It should be noted that this construct appears at the syntactic level of an equation, for its intuitive semantics is that of a set of equations (those of the subsystem).

Equation `::`
    Identifier `=` *Expression* `;`
  | `use` `[` *ExtensionDomain* `]` *Identifier*
        `[` `[`*ParamAssignment* `]` `]`
        `(` *ExpressionList* `)`
    `returns` `(` *IdentifierList* `)` `;`

Why not give a subsystem `use` an *expression* syntax instead of this *equation* syntax, so that it may appear in any expression just as a function call in usual languages? A simple answer is that it would need a major change in the syntax in the case when a system has several output variables, as `FullAdder`: we would have to introduce list of expressions, making the syntax much more complex as well as the domain analysis. A deeper reason is that it would be very difficult to replace such an expression subsystem call with its definition: the subsystem is basically a set of equations, with a fixed point semantics which cannot always be expressed as an expression.

2.2.4 *Giving values to the subsystem's size parameters.* In Prog. 7 the subsystem `FullAdder` has no parameter. In the general case, the parameters of the subsystem need to be given a value, which may be any affine function of the parameters of the caller and the extension indices.

This feature is best explained using some examples. Consider the binary multiplication algorithm, which is very similar to the decimal usual algorithm performed "by hand". Fig. 4 shows that such a multiplication is basically a collection of additions.

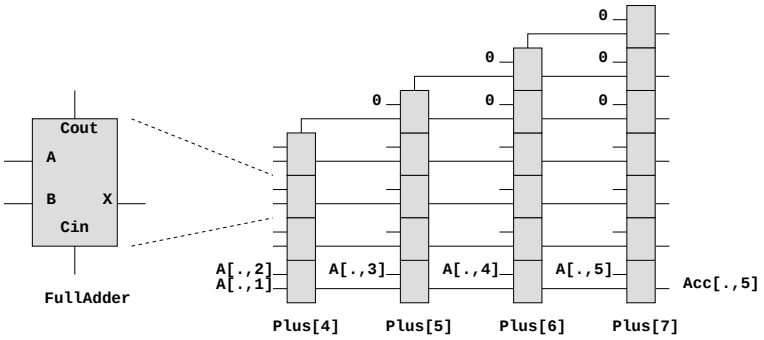


Fig. 5. Binary numbers accumulation.

Prog. 8 is the ALPHA incarnation of Fig. 4. Line 8 performs the binary product of all the bits of the first operand by each bit of the second. Line 10 describes a linear collection of additions, indexed by `m` which is the row index in Fig. 4. Lines 12-17 link the result of one additions to the input of the following.

```
1       system Times: {W|W>2} (A,B: {b| 0<=b<W} of boolean)
2                     returns (X  : {b| 0<=b<W} of boolean);
3       var
4         P : {b,m| 0<=b,m<W } of boolean;
5         Si : {b,m| 0<=b<W; 0<m<W } of boolean;
6         So : {b,m| 0<=b<=W; 0<m<W } of boolean;
7       let
8         P[b,m] = A[b] and B[m];
9
10        use {m| 0<m<W} Plus[W] (Si,P) returns (So);
11
12        Si[b,m] =
13          case
14            {| m=1} : P[b,m-1];
15            {| m>1} : So[b+1,m-1];
16          esac;
17        X[b] = So[b+1,W-1];
18      tel;
```

Program **8:** Binary multiplication in ALPHA

Here the parameter assignment `W` just equates the bit size parameter of the subsystem `Plus` and that of the multiplication. In the general case, however, this parameter assignment may be any affine function: Fig. 5 addresses for example the problem of accumulating binary numbers without overflow: the sum of two N-bits numbers may be a $N+1$ bits number, therefore it is needed to use additions of increasing sizes to avoid overflows.

Program 9 exhibits a `use` construct which corresponds to this figure. Another example where the "natural" structuring of an algorithm makes use of a param-

eter assignment depending on the extension indices is the Gaussian elimination [de Dinechin and Robert 1996].

```
use {i| 1<=i<N} Plus[W+i-1] (A1,A2) returns (Acc);
```

Program **9:** Parameter assignment depending on extension index

In all our examples, the extension domain is mono-dimensional, but in the general case the extension domain may be any arbitrary ALPHA domain. A typical example where it is useful is, in an image processing application, applying some function to all the points of an image. Another detailed application will be demonstrated in 3.5.

2.2.5 *A substitution semantics.* The semantics of a subsystem `use` is more subtle than it seems. The main difference with a more conventional function or procedure call is that the `use` expresses a set of computations which *is not monolithic*: line 10 of Prog. 8 hides a two-dimensional collection of instances of full-adder functions without implying any order on the evaluation or execution of these instances. In particular, all the inputs need not being defined on the whole of their domains for the `use` to have a semantic. In this program it wouldn't be possible, since the input `Si` is defined as a function of the output `So`. From this point of view, the `use` is to the procedure call what the recurrence equation is to the affectation.

For this reason, it is difficult to give a compositional semantics to the `use` (the problem is addressed in [Dupont de Dinechin 1997]). It is much easier and intuitive to define this semantics by substitution: a program containing a `use` is equivalent to a program in which this `use` has been replaced with the equations of the subsystem, plus equations for actual/formal input/output linking. These equations are usually deeply modified because of dimension extension, but the modifications are based on the usual and well-defined domain computations. This substitution semantics is detailed in [de Dinechin et al. 1995] and has been implemented as the `inline[]` program transformation in the MMALPHA environment.

3. MANIPULATING STRUCTURED SARES

Thanks to the `inline[]` transformation, any SSARE may be transformed into an equivalent SARE by inlining recursively all of its subsystems, until there is no more `use` instruction. Therefore any of the already existing tools, transforming a SARE into an equivalent one, may be straightforwardly extended to a tool transforming a SSARE into an equivalent SARE. This ensures that the existing MMALPHA environment may be used as well for algorithms expressed initially as SSAREs.

However, this approach obviously discards all the structure information, which could otherwise be usefully exploited. This section explores and demonstrates more clever uses of the SSAREs in MMALPHA.

3.1 Unchanged tools

First, the global inlining approach is sometimes uselessly brutal, for some of the tools need only minor change to adapt to structured SAREs. For example, the *localization* of a dependency is intrinsic to a given system, and may therefore be extended very straightforwardly to handle `use` statements as well as equations.

Such are also the *change of basis* transformation and the serialization of `reduce` constructs, among others.

3.2 Tools with new features

In general, however, handling structures involves major changes in a given transformation, hopefully offering new opportunities.

Consider for example the chain of transformation leading from an ALPHA SARE to a sequential C program. Presently the C program is a single-assignment one, and is therefore very memory inefficient. The translation to a sequential program with memory reuse will be eased by the ability to manipulate structures (interpreted, in this case, as sequential procedures). Of course none of the involved computations absolutely needs SSARE, but they help making interpretations explicit.

Another complex example is the *scheduling* of a SSARE, which may be done at several levels: one one side, there is a global scheduling of a SSARE, which is that of the equivalent, completely inlined one. On the other side, it is possible to consider some subsystems as *atomic*, and schedule a SSARE accordingly. Such a scheduling is needed for the previous problem of memory reuse in sequential code. A third possibility is to schedule each system separately, inheriting constraints on the schedule from its subsystems and caller systems. Subtle problems arise in this case, and these questions are the subject of ongoing research.

3.3 New tools

Program structures also open a range of new possibilities in the classical SARE-based program transformation environments. In the ALPHA project, a fruitful example is the definition of the ALPHARD language [Le Moënner et al. 1996], a subset of ALPHA which is straightforwardly interpretable as a hardware description, while still subject to strong automatic analysis. In this context, this analysis ensures the completeness and consistency of the connections and timings, making developing a parallel synchronous architecture easier in ALPHARD than in a standard hardware description language such as VHDL (an input to most silicon compilers). ALPHARD, however, is translatable into structural VHDL. The current MMALPHA environment also provides the translation into ALPHARD of a scheduled ALPHA program such as Prog. 3, and so ALPHARD is a convenient intermediate language for the synthesis of this still abstract architecture.

This intermediate language is based on structured SAREs: an ALPHARD description is strongly structured into boards, each board consisting of several modules, each module being a regular collection of elementary cells (a `use` with an extension domain defining this collection). Although possible without parameters and structures, such a description would be much less convenient, and its translation to VHDL would be much more difficult (VHDL is structured). The translation of scheduled ALPHA to ALPHARD is also conceptually easy: it basically consists of decomposing the declaration domains of computation variable into space domains and time domains, and then making cells with the time domains, and `use` extension domains in the modules from the space domains. Of course the actual transformations involved are much more complicated and beyond the scope of this paper.

3.4 New applications (1): specialization of parallel code

Although most of our examples so far are taken from the VLSI world, SARE-based formalisms are also useful in the connex fields of parallel programming and automatic parallelization. A recent example is the use of structured SAREs for the specialization of parts of the ScaLaPack library.

ScaLaPack provides portable parallel routines for most linear algebra algorithms on block-partitioned dense matrices. Sophie Robert used MMALPHA to optimize some of these routines (including LU decomposition) for Hessenberg matrices, which are triangular plus one lower diagonal. The idea was to remove all the computations involving null values in such matrices.

The procedure was the following:

—first translate a hierarchy of ScaLaPack functions into an equivalent hierarchy of ALPHA systems (using the structures defined in this paper),

—then cut out the null polyhedra (i.e. the triangle of zeroes below the diagonal) from the input matrices of these systems,

—then propagate, using MMALPHA, the corresponding information thanks to polyhedra computations and more subtle inference techniques,

—and finally translate back the resulting hierarchy into the original ScaLaPack code. Thanks to the hierarchical nature of this library, only parameters of function calls need changing.

These techniques are presented in detail elsewhere [de Dinechin et al. 1997]. In the case of the LU decomposition they allowed an improvement of complexity from $O(n^3)$ to $O(n^2)$ in very short time.

3.5 New applications (2): schedule-free component libraries

We address here the synthesis of *digital* circuits, that is circuits computing numerical values. The design flow previously sketched in 1.5 yields parallel architectures whose granularity is that of a numerical value. However, these values are implemented as fields of bits (words), and there is often some potential parallelism at the granularity of the bit which remains unexploited. Exploiting this bit-level parallelism may lead to architectures with bit-level pipelines, which may be either more efficient for a given silicon cost (bit-parallel architectures), or of much lower silicon cost with roughly the same computational power (bit-serial architectures).

There are technological reasons which make such architectures difficult to synthesize, such as in some cases a higher clock frequency for the same computing power. We believe, however, that the main obstacle to bit-parallel architecture synthesis is simply the added degree of complexity of managing this parallelism: one has to rely on *libraries* of arithmetic operators to lower development time (e.g. datapath libraries), and it is these libraries that impose a word granularity.

We therefore introduce the concept of *schedule-free operator libraries:* a schedule-free operator is simply a system of affine recurrence equations (SARE) describing an arithmetic operation at the most abstract level. For example, Prog. 8 is a schedule-free multiplier, because it does not impose an order on the various bit-level computation involved, but for the implicit order due to data dependencies. The originality of this approach is thus that this order or schedule is not cast in

silicon when designing the operator, as it would be if we were designing a silicon (e.g. datapath) library. It will depend on the application in which the operator is to be used, and so will be determined later thanks to program analysis in this context.

The design process is then the following: a specification is written, simulated and validated at the word level, *i.e.* using abstract data-types (real or integer). An example of such a word-level specification is our matrix-vector product of Prog. 4. Then an automatic program transformation refines it into a functionally equivalent bit-level SARE, using a library of schedule-free arithmetic operators. The resulting bit-level SSARE then undergoes the synthesis process sketched in 1.5, yielding a regular array of bit-level processors.

The quality of the resulting circuits when implemented in silicon has been discussed elsewhere [de Dinechin 1997]. This method is mostly suited to programming Field Programmable Gate Arrays (FPGA), which consist of a two dimensional grid of programmable gates which can host our bit-level processors. We will however only focus here on the language aspects.

3.5.1 *Bit-level refinement transformation.* We call bit-level refinement the automatic program transformation that replaces, in a word-level specification like Prog. 1, the operators `+` and `*` with instances of the corresponding ALPHA SAREs, Prog. 7 and 8. The `real` variables need replacing with `boolean` ones, with one more dimension holding the bit representation.

This transformation is detailed in [de Dinechin 1997]. Its core is the computation of the *domains* of the operators, using automatic analysis tools as described in the first section. For example consider the following equation:

```
var C,M: {i,j| 1<=i<=N; 1<=j<=N};
...
    C = case
            ...
            {i,j | i=j} : M + C.(i,j->i-1,j) ;
        esac;
```

In this case the domain of the `+` – the set of additions described in this line – is computed by translating the domain of `C` thanks to the `(i,j->i-1,j)` translation, then intersecting it with the domain of `M`, then intersecting the result with the domain of `C` itself intersected with the `{i,j| i=j}` restriction domain. These domain computations are all done automatically [de Dinechin and Robert 1996].

Once this domain D_+ of the `+` operator is computed, we change all the `real` variables into `boolean` ones, and we replace this operator with a `use` of the subsystem `Plus`, *whose extension domain is exactly* D_+. Thus the resulting SSARE describes exactly the same set of additions, but now at the bit-level.

It is here obvious that a more conventional program structure, say a *map* on a list of arguments, would be inadequate, for it wouldn't allow for the expression of a collection of `+` of arbitrary polyhedral shape.

3.5.2 *Synthesis of the bit-level architecture.* The synthesis of the transformed program is then similar as in section 1.5. Note, however, that here the scheduling takes into account the dependencies at the bit level, therefore the operators are

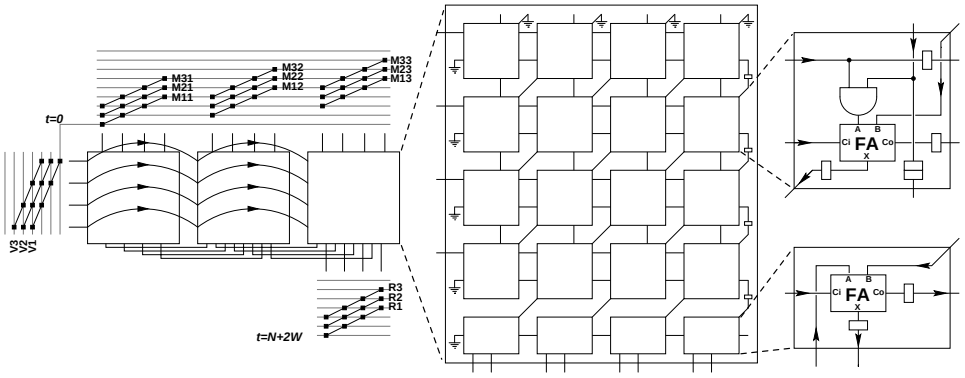


Fig. 6. Bit-parallel systolic array for the matrix-vector product

pipelined at the bit level. Besides, we still have at this point a parameterized design where we may vary the word size `W` and the parameters of the problem (here `N`, the size of the matrix).

In the case of the matrix-vector product, we get a regular processor array depicted on Fig. 6. The figure shows the bit arrays for each matrix or vector element, input and output in a *parallel skewed* manner (the isochrone lines are also shown).

Using similar techniques, other trade-offs will be possible, yielding for example a bit-serial circuit with a latency linearly worse than that of this bit-parallel array, but a linearly better silicon cost. This possibility is just evoked [de Dinechin 1997], since its automation has not yet been studied in depth.

Conclusions and future works

This paper introduces structured systems of affine recurrence equations in the polyhedral model, and their use and applications in the Alpha language and the MmAlpha environment. These extensions to a classical model allow to use it for non-trivial tasks, some of which are demonstrated in the field of VLSI synthesis and parallel programming. These extensions are based on a very general property of the set of array domains, which is its closure under a certain range of operations. Therefore, finer refinements of the polyhedral SARE model which preserve this closure property (replacing polyhedra with linearly bounded lattices [Teich and Thiele 1993] or $\mathbb{Z}$-polyhedra [Quinton et al. 1996]) will benefit of this work with only minor modifications.

Structured SAREs do not increase the power of expression of the language, just as any program written in an object-oriented language could also be written in assembler. We have shown, however, that most of the individual tasks involved in the program refinement approach are made easier by the availability of SARE structures: initial specification, program analysis, program transformation, translation of the final program into an implementation-specific target language. It also opens new opportunities, the examples given in this paper being the easy specialization of hierarchical parallel code or the use of schedule-free operator libraries. Several other applications exist, some of which have been studied thoroughly and implemented, some are under implementation, and some other are only evoked and

still in the domain of future research.

REFERENCES

Darte, A. and Robert, Y. 1994. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems 5*, 814–822.

de Dinechin, F. 1997. Libraries of schedule-free operators in Alpha. In *Application Specific Array Processors*. IEEE Conmputer Society Press.

de Dinechin, F., Quinton, P., and Risset, T. 1995. Structuration of the Alpha language. In *Massively Parallel Programming Models*, W. Giloi, S. Jahnichen, and B. Shriver, Eds. IEEE Computer Society Press, 18–24.

de Dinechin, F., Risset, T., and Robert, S. 1997. Hierarchical static analysis for improving the complexity of linear algebra algorithms. In *Parallel Computing*. Elsevier, Bonn, Germany.

de Dinechin, F. and Robert, S. 1996. Hierarchical static analysis of structured systems of affine recurrence equations. In *Application Specific Array Processors*. IEEE Computer Society Press.

Delosme, J. and Ipsen, I. 1986. Design methodology for systolic arrays. In *Proc. SPIE 30th Ann. Int. Tech. Symp. on Optical and Optoelectronic Applied Sciences and Engineering*. San Diego (USA), 245–59.

Dezan, C. 1993. Thèse de doctorat. Ph.D. thesis, université de Rennes I.

Dezan, C., Gautrin, E., Le Verge, H., Quinton, P., and Saouter, Y. 1991. Synthesis of systolic arrays by equation transformations. In *Application Specific Array Processors*. IEEE, Barcelona, Spain.

Dupont de Dinechin, F. 1997. Thèse de doctorat. Ph.D. thesis, université de Rennes I.

Feautrier, P. 1992. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *International Journal of Parallel Programming 21,* 5 (Oct.).

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

Kung, H. and Leiserson, C. 1978. Systolic arrays (for VLSI). In *Sparse Matrix Proc. 1978*. Society for Industrial and Applied Mathematics, 256–282.

Kung, H. T. January 1982. Why systolic architectures? *IEEE Computer 15,* 1, 37–46.

Le Moënner, P., Perraudeau, L., Rajopadhye, S., Risset, T., and Quinton, P. 1996. Generating regular arithmetic circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*. 429–436.

Moldovan, D. 1982. On the analysis and synthesis of VLSI algorithms. *IEEE Transactions on Computers C-31,* 11 (Nov.).

Quinton, P. 1984. Automatic synthesis of systolic arrays from recurrent uniform equations. In *11th Annual Int. Symp. Computer Arch., Ann Arbor*. 208–214.

Quinton, P., Rajopadhye, S., and Risset, T. 1996. Extension of the Alpha language to recurrences on sparse periodic domains. In *Application Specific Array Processors*. IEEE Conmputer Society Press.

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

Rajopadhye, S. and Fujimoto, R. 1987. Systolic array synthesis by static analysis of program dependencies. In *Parallel Architectures and Languages Europe*, J. de Bakker, A. Nijman, and P. Treleaven, Eds. Springer-Verlag, 295–310.

Rajopadhye, S. and Wilde, D. 1995. The naive execution of affine recurrence equations. In *Application Specific Array Processors*. IEEE Computer Society Press.

Rajopadhye, S. V., Purushothaman, S., and Fujimoto, R. M. 1986. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer Verlag, LNCS No 241, New Delhi, India, 488–503.

Rao, S. 1985. Regular iterative algorithms and their implementations on processor arrays. Ph.D. thesis, Standford University, U.S.A.

Teich, J. and Thiele, L. 1991. Control generation in the design of processor arrays. *Journal of VLSI Signal Processing 3*, 77–92.

Teich, J. and Thiele, L. 1993. Partitionning of processor arrays: a piecewise regular approach. *Integration, the VLSI Journal 14,* 3 (Feb.), 297–332.

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

Wilde, D. K. 1994. The Alpha language. Publication Interne 827, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. Jan.

Yaacoby, Y. and Cappello, P. R. 1988. Converting affine recurrence equations to quasi-uniform recurrence equations. In *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory*. Springer Verlag. See also, UCSB Technical Report TRCS87-18, February 1988.

Yang, J. and Choo, Y. 1991. Parallel-program transformation using a metalanguage. *ACM Sigplan Notices 26,* 7 (July), 11–20.