Designing Parallel Programs and Integrated Circuits

Patrice Quinton

IRISA – ENS Cachan, Campus de Ker Lann, 35170 Bruz, France
Patrice.Quinton@bretagne.ens−cachan.fr

Tanguy Risset

CITI, INSA−Lyon, Bât Léonard de Vinci, 21 av Jean Capelle, 69621 Villeurbanne Cedex, France
Tanguy.Risset@insa−lyon.fr

Katell Morin−Allory

TIMA, 46 av Félix Viallet, 38031 Grenoble Cedex, France
Katell.Morin@imag.fr

David Cachera

IRISA – ENS Cachan, Campus de Ker Lann, 35170 Bruz, France
David.Cachera@bretagne.ens−cachan.fr

The design of parallel programs or architectures is of utmost importance in the context of todays integrated circuits where hundreds of processing units can be assembled to achieve specific tasks. We present our effort to develop methods and tools to solve this problem in the context of the *polyhedral model* where calculations are expressed as operations on polyhedra−shaped collections of data. These ideas are the basis of our MMAlpha toolbox that is implemented using *Mathematica*. We present MMAlpha and we explain how it can be used to design parallel architectures and to prove properties of such architectures.

■ 0. Introduction

Designing parallel programs has always been a challenge of computer science, and it is becoming more and more interesting, as even common processors tend to use several processors (see http://en.wikipedia.org/wiki/Multicore). Moreover, the development of embedded devices including high−performance signal processing algorithms calls for parallel architectures, for example, in third generation mobile telephones.

For more than 15 years, some of the authors (P. Quinton and T. Risset) have been involved in research regarding parallel integrated circuits. Basic mathematical ingredients of this research are program analysis, combinatorial optimization, polyhedral theory, and linear algebra. The need to develop tools and capitalize on these tools in order to exchange ideas among researchers lead them to find out a programming framework that would be flexible and powerful enough to host results of this research.

Using *Mathematica*, they developed together with other researchers, a framework called MMAlpha. This framework contains a few tens of *Mathematica* packages allowing

parallelism to be extracted from programs using various transformations. The input of MMAlpha is a program written in a single−assignment language called Alpha. In Alpha, a program is written as a set of recurrence equations, such as for example:

$$y_i = \sum_{k=0}^{K} w_k \, x_{i-k} \qquad (1)$$

where x_i is an input signal, w_k is a vector of coefficients, and y_i is the output signal (this equation defines a finite impulse response filter). By applying to the textual description of such a program a set of transformations, one can derive a parallel architecture for its execution, by observing that the behavior of a synchronous processor can also be described using recurrence equations over time and space. This is just what is implemented in MMAlpha.

In this paper, we present our effort to implement this research using *Mathematica*. Section 1 explains what MMAlpha is. Section 2 gives more details about how it is implemented using *Mathematica.* In Section 3, we explain how we use MMAlpha to prove properties of parallel circuits. Section 4 concludes the paper.

■ 1. What is MMAlpha ?

MMAlpha is based on a single−assignment equational language named Alpha that we introduce in section 1.1. Section 1.2. describes MMAlpha, our toolbox to transform Alpha programs into parallel architectures. In Section 1.3., we briefly develop an example of finite impulse filter. Section 1.4. gives some references on the polyhedral model.

□ 1.1. Alpha and the Polyhedral Model

Alpha is a programming language based on recurrence equations. Recurrence equations are widely used in mathematics and signal processing to express usual calculations. In the following, we shall use the example of a Finite Impulse Response (FIR) Filter:

$$y_i = \sum_{k=0}^{K} w_k \, x_{i-k} \qquad (2)$$

The corresponding Alpha program is as follows

```
system fir :   {N | 3<=N}
               (x : {i | 0<=i} of integer[S,16];
                w : {n | 0<=n<=N-1} of integer[S,16])
       returns (y : {i | N-1<=i} of integer[S,16]);
var
  Y : {i,n | N-1<=i; -1<=n<=N-1} of integer[S,32];
let
  Y[i,n] =
      case
        { | n=-1} : 0[];
        { | 0<=n} : Y[i,n-1] + w[n] * x[i-n];
      esac;
  y[i] = Y[i,N-1];
tel;
```

This program (called a *system*) is parameterized by a size parameter N whose value is constrained by the inequality $\{N \mid 3 \le N\}$. Inputs of the system are variables x and w. Variable x is an infinite sequence of integer values that represents the input of the filter. Variable w contains the coefficients of the filter: it is a finite sequence of integer values

of size N. The results of this system is the variable y, again an infinite sequence of integers. All variables have the type integer[S, 16] which means 16−bit signed integers. System fir contains two recurrence equations. The first one defines a local variable Y, a 2−dimensional array. Its first dimension is the same as that of y, and its second dimension the same as x. $Y[i, n]$ is defined by a case expressions. When $n = -1$, $Y[i, n]$ takes value 0 (we shall explain the 0[] notation later on). When $n \geqslant 0$, $Y[i, n-1]$ takes the value $Y[i, n-1] + w[n] \times x[i-n]$. The second equation defines the output y of the system: $y[i]$ is the ultimate value of the sequence $\{Y[i, n]\}_{0 \leq n \leq N-1}$.

More generally, each Alpha variable is a collection of values associated with integral points of a polyhedron called the *domain* of this variable. Scalar variables such as the constant 0 above, are considered 0−dimensional array and addressed with `[]`. Alpha expressions all have a domain, whose definition depends on the way the expression is defined. For example, adding two expressions such as in $A + B$ gives an expression whose domain is the intersection of the domains of A and B.

The body of the program (between the let and tel keywords) express the recurrence relations. The `case` operator allows one to assign different values to expressions, depending on domain conditions. The restriction operator, denoted :, allows one to restrict the domain of an expression. For example, dom : exp restricts exp to the domain dom.

In Alpha, each expression has a domain, for instance the domain of expression $Y[i, n-1]$ is: {i , n | 1 <= i <= N + 1}, which corresponds to the pre−image of the domain of Y by the dependence function $(i, n \to i, n-1)$. The original Alpha notation for $Y[i, n-1]$ is $Y.(i, n \to i, n-1)$, but it is usually more convenient to represent Alpha programs in *array notation* as it is done above.

Alpha programs enjoy the *single assignment property*: each array element of an Alpha variable is defined exactly once. Related to that, there is no predefined order of execution for the computation of the program: an Alpha system is therefore a definition rather than an algorithm. The order of execution will be determined by the scheduling process detailed in the next paragraph. Because of this single assignment property, an Alpha program contains only true dependences between computations: dependence analysis is greatly simplified and parallelism detection is easier.

Because of its restricted syntax, Alpha is well−suited for source−to−source program transformation. The Alpha language has been designed to ease the program transformation in order to, starting from a functional specification such as the fir system, refine the specification down to a precise operational implementation of the same specification. The transformations of Alpha programs are performed thanks to transformations implemented in the MMAlpha programming environment.

□ 1.2. MMAlpha

MMAlpha is the programming environment used to manipulate and transform Alpha programs. MMAlpha consists in a set of *Mathematica* packages interfaced with C libraries for parsing Alpha programs and doing computations on polyhedra.

From the point of view of a chip designer, MMAlpha can be seen as a High Level Synthesis (HLS) software: the user transforms an algorithmic description − an Alpha program − into a digital circuit implementing this description using a Hardware Description Language (HDL) such as VHDL or SystemC. MMAlpha is not supposed to do the refinement automatically, it only provides a toolbox to ease the transformations of Alpha programs.

Mathematica manipulates internal representations of Alpha programs. These internal representations, called Abstract Syntax Trees (AST), are *Mathematica* symbolic expressions. The MMAlpha packages provide analysis and transformations of AST. The user loads an Alpha program, performs various transformations, eventually save the transformed program and translates the final result into a HDL. At any time, the user can simulate the Alpha program by generating C code and executing it.

In general, the transformation process consists in the three main following steps, once the program is loaded:

- Schedule the program: finds out when the calculation of some variable instance, say $Y[i, n]$ is computed. This is done automatically using one of the available schedulers of MMAlpha.
- Place the computations: once a schedule has been found, rewrite the program in such a way that variables now have time and space indexes. For example, $Y[i, n]$ is replaced by $Y[t, p]$ where t represents the time at which, and p the processor number where this expression is computed. A placed system can therefore be interpreted as an architecture.
- Transform to hardware description language: from the skeleton produced by schedule and place, derive a *real* architecture in a Hardware Description Language.

To illustrate the use of MMAlpha, we provide the derivation that a user, usually a hardware designer, will execute on a simple example: the FIR filter

1.3. Example of the FIR filter

Consider the program given in section 1.1. A hardware description executing such a program can be obtained using the following *Mathematica* program:

```
In[99]:= (* 1 *) load["fir.alpha"];
         (* 2 *) scd[addConstraints → {TxD1 == 1, TyD1 == 1},
           optimizationType → Null];
         (* 3 *) appSched[];
         (* 4 *) toAlpha0v2[]; alpha0ToAlphard[];
         (* 5 *) fixParameter["N", 8];
         (* 6 *)a2v[];
```

Commands are explained as follows:

1. Parse and load the Alpha program.
2. Schedule it. Some constraints and options are given to the scheduler in order to reach a *good* solution.
3. Place calculations using the schedule.
4. Translate program into a hardware description, still using Alpha syntax.
5. Set the value of the N parameter.
6. Translate to VHDL.

The following presents a small part of the resulting VHDL file:

```
-- VHDL Model Created for "system cellfirModule1"
-- 31/3/2006 7:44:9.477505
-- Alpha2Vhdl Version 0.9

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

library work;
use work.definition.all;


ENTITY cellfirModule1 IS
PORT(
  clk: IN STD_LOGIC;
  CE : IN STD_LOGIC;
  Rst : IN STD_LOGIC;
  wXMirr1 : IN  SIGNED (15 DOWNTO 0);
  xXMirr1 : IN  SIGNED (15 DOWNTO 0);
  YReg1Xloc : IN  SIGNED (31 DOWNTO 0);
  Y : OUT  SIGNED (31 DOWNTO 0)
);
END cellfirModule1;

ARCHITECTURE behavioural OF cellfirModule1 IS
    SIGNAL YReg1 :  SIGNED (31 DOWNTO
  0) := "00000000000000000000000000000000";


  -- Insert missing components here!---------
BEGIN

    PROCESS(clk) BEGIN IF (clk = '1' AND clk'EVENT) THEN
      IF CE='1' THEN YReg1 <= YReg1Xloc; END IF;
                END IF;
    END PROCESS;

    Y <= (YReg1 + (wXMirr1 * xXMirr1));

END behavioural;
```

For the reader who is not familiar with the VHDL language, the few final lines (between the BEGIN and the END behavioural keywords) describe a register (as a VHDL PROCESS) and a multiply and add operator that constitute the basic element of this architecture.

□ 1.4. The Polyhedral Model

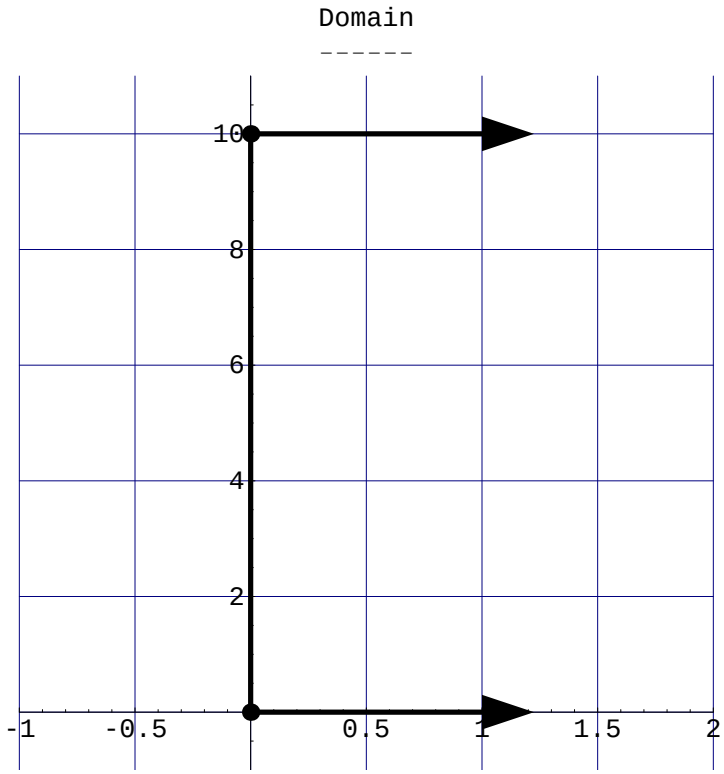
MMAlpha is based on the so−called *polyhedral model* of computation where computations are expressed as operations on data collections associated with integral points of polyhedra. Polyhedra are an interesting structure from the algebraic point of view, as they are stable under many interesting operations: intersection, (convex) union, pre−image by an affine function. (In fact, the data structure that used in Alpha is the finite union of polyhedra, which is also stable by union.)

Finding out the optimum of a linear form on a polyhedron is a well−known problem tackled by linear programming methods. In MMAlpha, polyhedra are represented by their dual representation [Schrijver86]. A polyhedron is either represented by a finite set of inequalities or by its system of generators including its vertices and rays (directions of half−lines contained in the polyhedron).

For example, the polyhedron $\{i, j \mid 0 \le i \wedge 0 \le j \le 10\}$ has two vertices $v_1 = (0, 10)$ and $v_2 = (0, 10)$ and one ray $r = (1, 0)$ as shown here. It can be represented either as a set of

inqualities $\{0 \le i \wedge 0 \le j \wedge j \le 10\}$ or as the sum of a convex combination of its vertices and of a positive combination of its rays $\sum_{0 \le \alpha \le 1} \alpha v_1 + (1-\alpha) v_2 + \rho r$.

```
In[33]:= vshow[readDom["{i,j|0<=i;0<=j<=10}"]]
```




```
Out[33]= - Graphics -
```

This representation is well−suited to the analysis of loops in imperative languages, as the iteration space of a loop, that is to say, the range of the loop indexes, is usually a polyhedron. Therefore, it is quite natural to represent loop statements as a single operation on sets of values indexed by the iteration indexes.

The use of recurrence equations to model parallel computations was first described in [Karp67]. Leslie Lamport [Lamport74] was the first to model loop parallelization for parallelization. Later on [Moldovan83] and [Quinton84] developed techniques to represent parallel architectures using recurrence equations and dependence analysis. Feautrier [Feautrier86] contributed notably to the polyhedral model in various way, among which the invention of parameterized integer programming. The definition of the Alpha language was due to Mauras [Mauras89].

■ 2. The Implementation of MMAlpha

In this section, we briefly describe how MMAlpha is implemented using *Mathematica*.

□ 2.1. Packages and external libraries

MMAlpha is implemented as a set of about 50 *Mathematica* packages that describe all the transformations. The software is freely available (http://www.irisa.fr/cosi/ALPHA/welcome.html).

These *Mathematica* packages make use of a few external C programs that perform calculations that would be inefficient if written in *Mathematica* or that were easier to implement in an imperative language.

The *Polylib* library is such a basic external tool: it allows operations on polyhedra to be done efficiently. Polylib is interfaced to MMAlpha using *Domlib*, a *Mathematica* Package that uses Mathlink to call the Polylib programs.

Parsers and un−parsers of Alpha are also written in C.

Finally, MMAlpha uses another C tool named Pip (http://www.piplib.org/) that performs parametric integer programming[Feautrier86]. Pip allows one to find out the lexicographic minimum (or maximum) of a parameterized polyhedron.

□ 2.2. Programming Style and *Mathematica* features used

MMAlpha packages are mainly written using the functionnal style. As many MMAlpha functions are transformations of the abstract syntactic tree of the program, patterns are extensively used to locate parts of the tree that have to be transformed. A few packages are written directly as a set of replacement rules.

One main concern while writing programs using *Mathematica* was to find out a programming style that would be safe enough to be used for such a big application. We more or less adhered to principles presented in Roman Maeder's book [Maeder97].

The initial decision to use *Mathematica* (in 1989!) was based on the foreseen interest of using symbolic libraries and also rewriting systems. Actually, MMAlpha uses less possibilities than we expected. Still, MMAlpha uses linear solvers, some optimization tools, and some Graphics facilities.

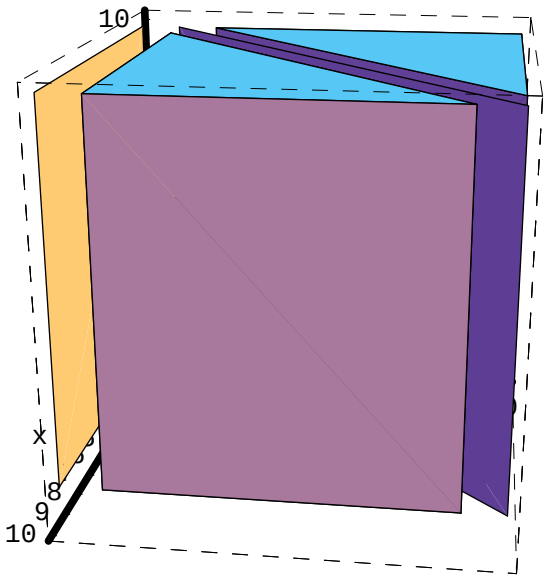
Mathematica has been found to be a very stable, always upwards compatible, and portable software. These are very interesting qualities for hosting a long−term research effort. Its main drawback is that is does not allow *Mathematica*−independent applications to be produced: this makes it difficult to draw the interest of researchers in communities where this software is not available.

Below is an example of using the graphics capabilities of *Mathematica* in order to vizualize domains. The following Alpha program defines a 3−dimensional variable A whose domain is a cube of size 10. However, the definition of A is broken into four different regions.

```
system strange (a : {i,j | 1<=i<=10; 1<=j<=10} of integer)
       returns (c :  integer);
var
  A : {i,j,k | 1<=i<=10; 0<=j<=10; 1<=k<=10} of integer;
let
  A[i,j,k] =
      case
        { | j=0} : a[i,k];
        { | j+1<=i; 0<=j} : A[i,j-1,k];
        { | i<=j-1; 0<=j} : A[i,j-1,k];
        { | i=j} : a[i,i];
      esac;
tel;
```

The following MMAlpha expression allows one to create an animation to visualize the domain of variable A (open the cell, select the group of graphic cells and type command−y to animate it.)

```
vshow["A", minR → 0.2, stepR → 0.2, maxR → 3]
```




```
Out[31]= {-Graphics3D-, -Graphics3D-, -Graphics3D-,
          -Graphics3D-, -Graphics3D-, -Graphics3D-, -Graphics3D-,
          -Graphics3D-, -Graphics3D-, -Graphics3D-, -Graphics3D-,
          -Graphics3D-, -Graphics3D-, -Graphics3D-, -Graphics3D-}
```

■ 3. Proving Properties of Alpha Programs

The use of systematic and semi−automatic rewritings together with the clean semantic basis provided by the polyhedral model should ensure the correctness of the final implementation. Nevertheless, the development of real−size systems results in a loss of confidence in the initial high−level specification. Moreover, interface and control signal generators are not certified, and hand−made optimizations are still performed to tune the final result. This calls for the development of a formal verification tool to (partially) certify low−level system descriptions before their final implementation.

We have developed proof methods[CacMor05, Mor04] for control properties of systems expressed in the polyhedral model. We thus only deal with boolean signals, but our systems are parameterized. Due to undecidability results, the verification of such systems is known to be hard, since no automatic tool can be used to formally check that a given implementation fulfills its specification. We chose to develop a specific logic and proof system for Alpha programs.

The proof methods have been implemented in MMAlpha, and make intensive use of MMAlpha features like polyhedra manipulations or rewriting of equations. These polyhedra manipulations allow us to *hide* inductive proofs behind syntactic substitutions. The main idea is to substitute variables by constant boolean values, and to propagate them. The analysis of dependencies is the key to detect patterns in the definition of variables, revealing hidden self−dependencies and thus enabling further substitutions,

until enough information has been computed on variable values to ensure the desired property.

□ 3.1. A Proof System

Properties of a polyhedral system are described in a so−called *polyhedral logic*: a formula of this logic has the form $D : e \downarrow v$, where D is a polyhedral domain, e a polyhedral multidimensional expression, and v a boolean scalar value meaning that on domain D, expression e has value v where v is either tt (true) or ff (false). A formula $D : e \downarrow v$ is satisfied in the context of a polyhedral system S iff the value of expression e on the whole domain D is v. Proof for such formulae are constructed by means of a set of inference rules, that are of two kinds: (i) *classical* propositional rules, and (ii) rules *specific* to the model, based on heuristics using rewritings and polyhedral computations (*e.g.* intersection of polyhedra). We will give here some examples of these inference rules.

Classical propositional rules

Axioms

They are of two kinds. For a formula $D : e \downarrow v$

– if D is empty then the formula is satisfied,

– if e is a boolean polyhedral constant, value of which is v then the formula is satisfied.

Dependency Rule

Let F be a formula defined by $D : e.d \downarrow v$, then F is satisfied if formula $d(D) : e \downarrow v$ is satisfied.

Conjunction splitting rule

Let F be a formula defined by $D : e \wedge g \downarrow$ tt, then obviously F is satisfied if formulae $D : e \downarrow$ tt and $D : g \downarrow$ tt are satisfied. A similar rule exists for the disjunction and the value *ff*. But this rule cannot be used on formulae like $D : e \vee g \downarrow$ tt, since it may be satisfied even if formulae $D : e \downarrow$ tt and $D : g \downarrow$ tt are not satisfied.

For this kind of formulae, no classical rules can be used, and we therefore developed specific rules. The implementation of classical rules offers no difficulty and will not be discussed here.

Specific Rules

Constant substitution

Let us assume that we want to prove a formula $D : X \downarrow v$ where X is defined on a domain D_X by expression e_X. The idea is to simplify all the system expressions by propagating boolean constant values along the dependencies appearing in the definition of X. Let e be an expression used in system S. Expression e is simplified by substituting a given set of occurrences of X in e by the boolean constant value v. Let us assume that $X.d$ is an occurrence of X in expression e. The substitution of $X.d$ is not done on the whole domain D_X but on a subdomain which is the pre−image of domain D by dependency d.

The implementation of this rule is mainly based on the *Mathematica* functions **ReplacePart** and **Position**. For each branch $D_i : e_i$ of each variable definition, we extract the position of the dependency expressions dealing with X (expressions of the form $X.d$). These expressions are defined on the domain D_i, and they can be substituted only if they are defined on the domain D of the formula. We select only those which can be

substituted (the selection is done by computing a pre−image, an intersection and testing the emptiness of this intersection) and we substitute them by v.

Pattern detection

The constant substitution rule can only substitute instances of a variable by a constant boolean value. The idea of pattern detection is to substitute a whole expression by a variable, in order to use the substitution rule afterwards. Let us consider a variable P defined by an expression $P = X \vee Y$, which will be called a *pattern*. Let us assume that system S contains an expression $e = X.d \vee Y.d \vee Z.\delta$. We can recognize in e the pattern P composed with dependency d. Thus, we may substitute $X.d \vee Y.d$ in e by $P.d$. Since the pattern may be more sophisticated, the difficulty consists in finding the right dependency d. This requires solving a system of diophantine equations in $\mathbb{Z}^n$. This rule is implemented on a similar way than the constant substitution rule: the selection criteria is based on solving a diophantine system.

Self−dependency detection

This rule consists in rewriting a variable's expression defined without self−dependencies into an expression with self−dependencies. It thus detects *hidden* self−dependencies in the definition of variables, to allow further substitutions. This rule mainly relies on the pattern detection mechanism.

All these rules are sound with respect to the semantics. They are represented in *Mathematica* by functions. They are used in the proof algorithm to automatically attempt to prove formulae.

□ 3.2. Proof algorithm

Proofs are constructed in a classical manner, by building a proof tree according to the proof rules. Each node of the tree is labelled by a formula, the root being the initial formula we want to prove. At each node, a proof rule is automatically chosen, according to the structure of the attached formula. The children of a node are thus the premises of the chosen rule.

The construction stops in three cases:

− we can apply an axiom on the formula labelling a node (in this case this node is called a *closed leaf*);

− the polyhedral expression contained in the formula has already occurred in an ancestor of the node;

− no more proof rule can be applied.

In the last two cases, the corresponding nodes are called *pending leaves*. We prove in [Mor04] that the proof tree construction terminates. The main argument of the proof is that the construction cannot create an infinite number of formulae.

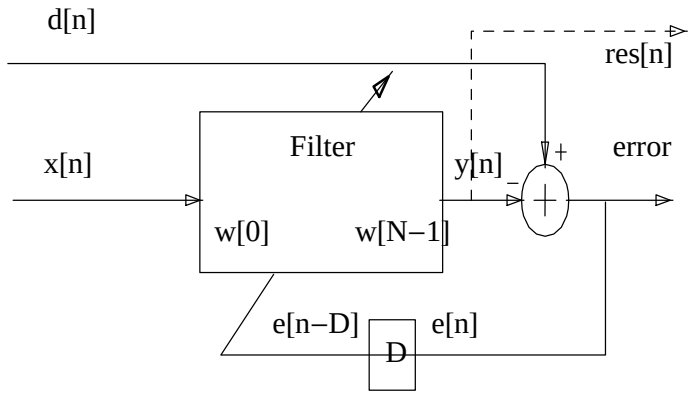
The proof of the formula is completed as soon as all the leaves are closed. But, in the general case, there are still pending leaves after the application of all possible rules.

Each node is represented in *Mathematica* by a list composed of a formula, a variable name from which the formula expression is extracted, and an integer used to tell what the next rule to apply is. The choice of the rule to apply is automatically done according to the syntactic tree of the formula expression.

□ 3.3. Applications

The algorithm has been used on several examples that we summarize briefly here.

An adaptive filter



This example deals with an adaptive filter [GQRM02] (see figure above). The system consists of N cells, each one containing a weight stored in a register. These cells are organized as a linear array. There are two inputs x and d, respectively for the raw input signal and for the desired output, and one output *res*. The system computes a convolution between input x and weights $w[i]$, then computes an error signal by subtracting the result by the desired output. This error is propagated backwards with a delay D to update weights. Under certain statistical assumptions on inputs, weights are converging to the desired values. This system is parameterized by parameters N, M, and D: N is the filter length (number of weights), D the delay and M the length of input flow x. In our example, due to the error feedback, some registers are not accessible as long as the output is not defined. An internal generated signal *Wctl1P* is used to control the access to registers $w[i]$. This signal is defined on the domain $\{t, p \mid t \leq p - N + M \wedge 0 \leq p \leq N - 1\}$, where parameters $N,\ M,\ D$ are defined on domain $\{N, M, D \mid 3 \leq N \leq \min(M - D - 1, D - 1)\}$. As long as the system is in its initialization phase, due to the feedback delay, weights are not correctly defined, so signal *Wctl1P* must be false in order to prevent access to these weights. In a second phase, this signal must become and stay true to allow access to the weight registers. Initial values of this signal are generated in the first unit, and then pipelined in the entire array. This is expressed by the equation displayed below.

```
Wctl1P[t, p] = {t, p | t ≤ D − 1; p = 0} : False
               {t, p | D ≤ t; p = 0} : True
               {t, p | 1 ≤ p} : Wctl1P[t − 1, p − 1]
```

Here we have two properties to prove:

$$\{t, p \mid p \leq t \leq p + D;\ 0 \leq p \leq N - 1\} : \text{Wctl1P}{\downarrow}\text{ff},$$

and

$$\{t, p \mid p + D \leq t < p - N + M;\ 0 \leq p \leq N - 1\} : \text{Wctl1P}{\downarrow}\text{tt}.$$

These two properties are proved in a similar way. Since *WctlP* depends on itself, the first rule that is applied is the constant substitution. For each property, the substitution is done on a different domain, and in both case the next rule to apply is an axiom. The proof construction leads only to closed leaves and the proof succeeds. All this process is fully automatic.

A matrix vector product

The proved property ensures that control signals are well defined. Presently this proof needs some user interaction since some heuristics need to be implemented. The proof of this property can be found in [CacMor05].

A hardware arbiter

We want to prove a mutual exclusion property. The proof is fully done in interaction with the user. We are improving our proof rules to decrease the number of user interactions [Mor04].

■ 4. Conclusion

We have described how parallelization techniques were implemented using *Mathematica* in the theoretical framework of the polyhedral model. The Alpha language that is used to represent both the specification of an application and its status during the transformation process was presented. Transformations were illustrated on the example of a finite impulse response filter. Finally, we have shown how properties of Alpha programs could be proved in the polyhedral framework.

Research on the polyhedral model is a long term effort whose goal is to develop a body of knowledge and algorithms that can be used in modern compilers or architecture design tools. Such an effort need be accompanied by the implementation of software programs that constitute building blocks for further development of the research. To this respect, *Mathematica* is an interesting and powerful platform that allows one to prototype new methods.

■ References

[CacMor05] D. Cachera and K. Morin−Allory, Verification of Safety Properties for Parameterized Regular Systems, *Transaction on Embedded Computing Systems, ACM* 4(2), pp 228−266, May 2005

[Feautrier88] P. Feautrier, Parametric Integer Programming, *RAIRO Recherche Opérationelle*, Vol. 22, Nr. 3, pp 243−268, 1988

[GQRM02] A.−C. Guillou, P. Quinton, T. Risset and D. Massicotte, Automatic Design of VLSI Pipelined LMS Architectures. In *Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS),* 2002

[Karp67] R.M. Karp, R.E. Miller, S. Winograd, The Organization of Computations for Uniform Recurrence Equations, *Journal of the ACM*, Vol. 14, Nr. 3, pp. 563−590, 1967

[Lamport74] L. Lamport, The Parallel Execution of Do−Loops, *Communications of the ACM*, Vol. 17, Nr. 2, pp. 83−93, 1974

[Mauras89] C. Mauras, Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones, Thèse de l'Université de Rennes 1, 1989

[Maeder97] R. Maeder, Programming in *Mathematica*, Addison−Wesley, 3d edition, 1997

[Moldovan82] D. I. Moldovan, On the Analysis and Synthesis of VLSI Algorithms, IEEE Trans. Computers, Vol. 31, Nr. 11, pp 1121−1126, 1982

[Mor04] K. Morin−Allory, Formal Verification in the Polyhedral Model, *PhD thesis,* Univ. Rennes 1, Oct. 2004 (in french)

[Quinton84] P. Quinton, Automatic Synthesis of Systolic Arrays from Recurrent Uniform Equations, *11th Annual Int. Symp. Computer Arch.*, pp. 208−214, Ann Arbor, June 1984

[Schrijver86] A. Schrijver, Theory of Linear and Integer Programming, Wiley, NewYork, 1986

■ Acknowledgment

Many people contributed to the implementation of MMAlpha (and of its predecessors, *Diastol* and *Alpha du Centaur*). Thanks to Catherine Dezan, Pierrick Gachet, Christophe Mauras, Hervé Le Verge, Zbignew Chamski, Sanjay Rajopadhye, Doran Wilde, Florent Dupont de Dinechin, Fabien Quilleré, Anne−Claire Guillou. Thanks also to all the students who contributed to the development of MMAlpha during internships.