Embedding of Systems of Affine Recurrence Equations in Coq

David Cachera and David Pichardie

IRISA / ENS Cachan (Bretagne)
IRISA, Campus de Beaulieu
F-35042 Rennes, France
{david.cachera,david.pichardie}@irisa.fr

Abstract. Systems of affine recurrence equations (SAREs) over polyhedral domains are widely used to model computation-intensive algorithms and to derive parallel code or hardware implementations. The development of complex SAREs for real-sized applications calls for the elaboration of formal verification techniques. As the systems we consider are generic, *i.e.*, depend on parameters whose value are not statically known, we considered using theorem provers, and have implemented a translation from SAREs into the Coq system. We take advantage of the regularity of our model to automatically generate an inductive type adapted to each particular system. This allows us to automatically prove that the functional translation of equations respects the wanted fixpoint properties, and to systematically derive mutual induction schemes.

1 Introduction

Systems of recurrence equations [10] provide a foundation for designing parallel multidimensional arrays for computation intensive applications. The polyhedral model uses an extension of the formalism of recurrence equations, specifically that of Systems of Affine Recurrence Equations (SAREs) over *polyhedral domains*. The ALPHA language [17] and the MMALPHA environment [16] provide the syntax and tools to handle such recurrence equations. The MMALPHA environment implements polyhedral transformations that allow one to refine a high level specification into a synthesizable architectural description. It also enables the derivation of imperative loop nest code.

Most of the transformations implemented in MMALPHA are rewritings that preserve the semantics, thus ensuring an equivalence between the original specification and the final code. Nevertheless, there are cases where we need to formally prove certain properties that are not ensured by the refinement process. For instance, the user is allowed to introduce hand-tuned optimizations during an intermediate refinement step, and we will have to check functional equivalence between the newly introduced fragment and the previous stage. Moreover, some properties we want to prove are not expressible in the formalism of SAREs. More generally, the development of more complex systems for real sized applications calls for the elaboration of verification tools. In particular, we have to be able to perform proof by induction following complex induction schemes.

The systems we consider are generic, *i.e.,* they depend on parameters whose value is not statically known. The proof assistant Coq provides higher-order logic and abstraction that are well-suited for our purposes. On the other hand, this kind of general-purpose theorem proving tool suffer from a lack of automation. User-defined tactics may partially overcome this problem, by automatizing proof steps which share a common structure.

The major contribution in this work lies in the fact that we make use of the specific features of the polyhedral model to generate a translation from recurrences equations to Coq.

- We first generate an intermediate functional translation of recurrence equation. Due to its type-theoretical nature, this function has to carry a proof of its termination.
- We automatically derive from this translation a simpler function, by just keeping the computational part that reflects the initial equations. To that purpose, we make use of a schedule of the system of equations given by the MMALPHA environment. This allows us to derive theorems that express system variables as solutions of fixpoint equations. These theorems will be the base for further proofs.
- At the same time, we automatically generate an induction principle that is specific to the considered system. Any inductive property about this system can thus be proved by instantiating this induction scheme. This strategy automatizes the most technical part of the proof (complex mutual recursions) and relies on the user only for some properties the prover cannot deal with (for instance, arithmetic properties that are not in a decidable fragment).
- We provide additional tactics that automate the use of fixpoint theorems by simplifying rewritings during the proof process.

The paper is organized as follows. Sections 2 present the polyhedral model. We show in section 3 how we translate SAREs into Coq, and in section 4 how we generate specific induction schemes and proof tactics. Section 5 gives two simple application examples. Section 6 is devoted to related work, and we conclude in section 7.

2 The Polyhedral Model

We now present the basic definitions of systems of recurrence equations and some related notions. In the following, we denote by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of, respectively, natural numbers, integers, rational and real numbers.

2.1 Polyhedral Domains and Recurrence Equations

Definition 1 (Polyhedral Domain). *An n-dimensional polyhedron is a subset $\mathcal{P}$ of $\mathbb{Q}^n$ bounded by a finite number of hyperplanes. We call* polyhedral domain *(or, for short,* domain*) a subset $\mathcal{D}$ of $\mathbb{Z}^n$ defined by*

$$\mathcal{D} = \mathbb{Z}^n \cap \mathcal{P}$$

where $\mathcal{P}$ is a finite union of n-dimensional polyhedra. A domain can be seen as a set of points, which will be called its indices.

Definition 2 (Variable). *An n-dimensional variable X is a function from a domain of $\mathbb{Z}^n$ into a base set (booleans, integers or reals). An* instance *of the variable X is a restriction of X to a single point z of its domain, and is denoted by $X[z]$. Constants are associated to the trivial domain $\mathbb{Z}^0$.*

Definition 3 (Recurrence Equations). *A* Recurrence Equation *defining a function (variable) X at all points, z, in a domain, $\mathcal{D}_X$, is an equation of the form*

$$X = \mathcal{D}_X : g(\ldots, X.d, \ldots)$$

where

- *X is an n-dimensional variable;*
- *d is a* dependency function*, $d : \mathbb{Z}^n \to \mathbb{Z}^n$; the value of expression $X.d$ at index i is the value of variable X at index $d(i)$;*
- *g is a* strict, single-valued *function; it is often written implicitly as an expression involving operands of the form $X.d$ combined with basic operators and parentheses;*
- *$\mathcal{D}_X$ is a polyhedral domain of $\mathbb{Z}^n$ and is called the* domain *of the equation.*

A variable may be defined by more than one equation. In this case, we use the syntax shown below:

$$X = \begin{cases} \texttt{case} \\ \qquad \mathcal{D}_{X_1} : g_1(\ldots X.d_{i_1} \ldots) \\ \qquad \vdots \\ \qquad \mathcal{D}_{X_l} : g_l(\ldots X.d_{i_l} \ldots) \end{cases} \tag{1}$$

Each line is called a branch, *and the domain of X is the union of the (disjoint) domains of all the branches, $\mathcal{D}_X = \bigcup_i \mathcal{D}_{X_i}$. We also say that the dependency function, d_i holds over the (sub) domain D_{X_i}.*

Definition 4. *A recurrence equation as defined above, is called an* Affine Recurrence Equation (ARE) *if every dependence function is of the form, $d(z) = Az + a$, where A is an integer $n \times n$ matrix and a is an integer n-vector.*

Definition 5. *A* system *of recurrence equations is a set of m such equations, defining the data variables $X_1 \ldots X_m$. Each variable, X_i is of dimension n_i, and since the equations may now be mutually recursive, the dependence functions f must now have the appropriate type.*

2.2 Dependence Graphs and Schedules

Definition 6 (Dependence Graph). *We say that an instance $X[z]$ of a variable X depends on an instance $Y[z']$ of a variable Y if there exists an equation*

of the form $X = g(\ldots, Y.d, \ldots)$ with $z' = d(z)$. We denote by $X[z] \to Y[z']$ this fact.

We call dependence graph the graph whose vertices are the variable instances, and where there is an edge between vertices x and y iff $x \to y$.

In order to get a valid implementation from a set of recurrence equations, we need (among other things) to determine a particular order in which computations should take place. The definitions above do not specify such an order, and do not even guarantee that it may exist.

Definition 7. *A schedule t_X is a function such that $t_X(z)$ specifies the time instant at which $X[z]$ is computed. Typically, the range of t_X is $\mathbb{N}$, but any total order is allowed (for example $\mathbb{N}^k$ under the lexicographic order). In the following, we restrict ourselves to one-dimensional schedules. A schedule is said to be valid if for any instances $X[z]$ and $Y[z']$,*

$$X[z] \to Y[z'] \implies t_X(z) > t_Y(z')$$

A set of recurrence equations is said to be schedulable if such a valid schedule exists. The problem of determining if a SARE is schedulable is undecidable in the general case [14], but much work has been devoted to develop heuristics to find schedules in specific cases.

2.3 The ALPHA language and the MMALPHA environment

The combination of polyhedral domains and affine dependence functions is the basis of the so-called *polyhedral model*. This model has a rich set of formal correctness preserving transformations. The ALPHA language, originally developed for the design of regular (systolic) arrays, provides a syntax to define and manipulate SAREs. We give here an example of an ALPHA program to get an insight on this syntax (see [17] for more details). The program displayed in Figure 1 implements an auto adaptive filter [11] used in signal processing for noise cancellation or system identification. It takes an input signal x and a reference input d and updates its convolution coefficients so as the output signal res will converge towards the reference input.

In this example, expressions such as {n | N<=n<=M} denote domains, N, M, D are parameters (respectively representing the number of weights of the filter, the number of inputs and the delay), x and d are the input variables, res is the output variable and W, Y, E are local variables. Note that a program may have several output variables.

This program is written in *array notation*: for syntactic convenience, instead of writing equations like W=...W.(n,i->n-1,i)... where (n,i->n-1,i) denotes the dependence function $(n, i) \mapsto (n-1, i)$, we use the following notation: W[n,i]=...W[n-1,i].... As a consequence, constants like 0 are followed by square brackets to indicate that they can be expanded from their initial domain $\mathbb{Z}^0$ to any multidimensional domain. Moreover, as in array notation indices are given a name in the lhs of equations, names indices do not appear any more in

```
system DLMS : {N,M,D | 3<=N<=(D-1,M-D-1)}
              (x : {n | 1<=n<=M} of integer;
               d : {n | N<=n<=M} of integer)
        returns (res : {n | N<=n<=M} of integer);
var
  W : {n,i | N<=n<=M; 0<=i<=N-1} of integer;
  Y : {n,i | N<=n<=M; -1<=i<=N-1} of integer;
  E : {n | N<=n<=M} of integer;
let
  W[n,i] =
       case
         { | n<=N+D-1} : 0[];
         { | N+D<=n} : W[n-1,i] +  (E[n-D] * x[n-i-D]);
       esac;
  Y[n,i] =
       case
         { | i=-1} : 0[];
         { | 0<=i} : Y[n,i-1] + (W[n,i] * x[n-i]);
       esac;
  E[n] = (d[n] - res[n]);
  res[n] = Y[n,N-1];
tel;
```

Fig. 1. DLMS ALPHA system

domain definition of case branches, hence the notation (for instance) {|N+D<=n} instead of {n,i|N+D<=n}.

The MMALPHA environment [16] implements a number of manipulations on ALPHA programs. The environment provides a set of predefined commands and functions to namely achieve the following purposes.

- Static analysis of programs, including analysis of polyhedral domains' shapes and sizes.
- Simulation, architectural description and VHDL generation.
- Transformations of ALPHA programs, based on polyhedra manipulations and including pipelining of variables, scheduling, change of basis, etc.

3 Translation of SAREs into Coq

To prove properties about a SARE, we first must translate it into the Coq system. We benefit from the equational nature of the underlying model to directly translate the constructs of our language into the formalism of Coq, instead of writing down a formalization of its semantics.

Each variable of a SARE is a partial function defined on a polyhedral domain. There are many ways of translating them to Coq. These methods reflect the semantics in a more or less precise manner. We follow an approach in two steps:

- Variables are first defined as mutually recursive functions. As we deal with a constructive logic, in addition to their indices, variables need an argument representing a proof that these indices are "accessible". This fact is explained below.
- The only information the user needs in a proof process is the set of equalities defining the system. We thus have to show that the variables previously defined are indeed a fixpoint for these equations. We will use a schedule to prove this fact by induction on time.

These steps are detailed in the next subsections.

3.1 Domains

Let us consider a variable X of domain $\mathcal{D}_X$, and of dimension n. This domain is simply represented by a predicate over $\mathbb{Z}^n$, holding a conjunction of affine constraints on domain indices. This results in the following type.

$$\mathrm{Dom}_X : \overbrace{\mathtt{Z} \; \mathtt{->} \; \ldots \; \mathtt{->} \; \mathtt{Z}}^{n \text{ times}} \; \mathtt{->} \; \mathtt{Prop}$$

3.2 Mutually recursive functions

Let us consider a SARE defined as follows, where each variable X_j is defined on a domain of dimension n_j and has a number l_j of case branches.

$$X_1 = \begin{cases} \mathtt{case} \\ \qquad \vdots \\ \quad \mathcal{D}_{X_{1,i}} : g_{1,i}(\ldots X_j.d_{1,i,j} \ldots) \\ \qquad \vdots \end{cases}$$

$$X_p = \begin{cases} \mathtt{case} \\ \qquad \vdots \\ \quad \mathcal{D}_{X_{p,i}} : g_{p,i}(\ldots X_j.d_{p,i,j} \ldots) \\ \qquad \vdots \end{cases}$$

We want to define these variables as mutually recursive functions. As we are in a type-theoretic framework, we must have a structurally decreasing argument in the definition of recursive functions. A well-known technique to address this point is to use an accessibility predicate [5]. The standard technique is to use a well-founded recursion principle derived from the accessibility predicate. We follow Bove's more specific approach which defines a special-purpose accessibility predicate for each function: in addition to its indices, each variable carries an argument stating that these indices are accessible. In this particular case, accessibility is defined by means of the dependencies between variable instances,

the base case being that of input variables whose indices are trivially accessible on their whole domain. This accessibility argument is structurally decreasing: if an instance $X[z]$ depends on an instance $Y[z']$ then the accessibility argument of $Y[z']$ is a subterm of that of $X[z]$.

```
Inductive Acc_{X_1} : Z → ··· → Z → Set :=
              | Acc_{X_{1,1}} : (i_1, …, i_{n_1} : Z)(Dom_{X_1} i_1 … i_{n_1}) →
                                    < D_{X_{1,1}} >→ (Acc_{X_1} i_1 … i_{n_1})
              /* case branch without dependencies */
              ⋮
              | Acc_{X_{1,l_1}} : (i_1, …, i_{n_1} : Z)(Dom_{X_1} i_1 … i_{n_1}) →< D_{X_{1,l_1}} >→
                      ··· → (Acc_{X_j}(d_{1,l_1,j} i_1 … i_{n_1})) → ··· → (Acc_{X_1} i_1 … i_{n_1})
              /* case branch where X_1 depends on some variables X_j*/
          ⋮
  with Acc_{X_p} : Z → ··· → Z → Set :=
              ⋮
              | Acc_{X_{p,i}} : (i_1, …, i_{n_p} : Z)(Dom_{X_p} i_1 … i_{n_p}) →< D_{X_{p,i}} >→
                      ··· → (Acc_{X_j}(d_{p,i,j} i_1 … i_{n_p})) → ··· → (Acc_{X_p} i_1 … i_{n_p})
          ⋮
```

Fig. 2. Accessibility predicate in Coq-like notation

We begin with the definition of an accessibility inductive type, simultaneously defined for all variables. This inductive type is displayed on Figure 2. In this inductive type definition, there is one type constructor for each branch in the definition of each variable: constructor $Acc_{X_{j,i}}$ is used for the ith branch of the jth variable. Moreover, $< D_{X_{j,i}} >$ denotes the set of affine constraints involved by domain $D_{X_{j,i}}$.

Using this accessibility type, we now can define the variables as mutual fixpoints. The definition uses a filtering on an accessibility "witness", that is, a term expressing the fact that the considered indices are indeed accessible (cf. Figure 3).

3.3 Using a particular schedule

As such, variables are already defined as a fixpoint of the equations. But using this specification in a proof process presents the major drawback of handling this accessibility term in any proof step. To simplify the proof procedure, we aim at suppressing this argument. We rely on a particularity of the polyhedral model to do this. In the following, we use a schedule of the system to make a recurrence on a variable representing time. As a consequence, our technique only applies to schedulable systems. In practice, this restriction does not matter since only schedulable systems can be implemented.

```
Fixpoint X₁_f : [i₁, ..., i_{n₁} : Z][acc : (Acc_{X₁} i₁ ... i_{n₁})] : Z :=
    Cases acc of
    | (Acc_{X_{1,1}} i₁ ... i_{n₁} _ _) ⇒ g_{1,1}
      ⋮
    | (Acc_{X_{1,l₁}} i₁ ... i_{n₁} _ _ ... dep_j ...) ⇒ (g_{1,l₁} ... (X₁_f (d_{1,l₁,j} i₁ ... i_{n₁}) dep_j) ...)
with X₂_f : ...
```

Fig. 3. Definition of a SARE as a mutual fixpoint

Given a particular schedule, we will prove that for any variable X_j and for any indices $i_1, \ldots, i_{n_j}$, we have the following implication[1].

$$(\text{Dom}_{X_j}\ i_1 \ldots i_{n_j}) \Rightarrow (\text{Acc}_{X_j}\ i_1 \ldots i_{n_j})$$

The particular schedule we work with is given by MMALPHA scheduling heuristics, and is expressed as a function $\mathbb{Z}^{n_j} \to \mathbb{Z}$, denoted by $t_{X_j}[i_1, \ldots, i_{n_j}]$

The proof proceeds in two steps.

- We first prove by induction (in $\mathbb{Z}_+$) the following theorem

  Theorem 1 (Accessibility w.r.t. time).

  $$\forall t \geq 0\ (\forall (i_1 \ldots i_{n_1}) \in \mathcal{D}_{X_1}, \text{if } t = t_{X_1}[i_1, \ldots, i_{n_1}] \text{ then } (\text{Acc}_{X_1}\ i_1 \ldots i_{n_1}))$$
  $$\vdots$$
  $$\wedge\ (\forall (i_1 \ldots i_{n_p}) \in \mathcal{D}_{X_p}, \text{if } t = t_{X_p}[i_1, \ldots, i_{n_p}] \text{ then } (\text{Acc}_{X_1}\ i_1 \ldots i_{n_p}))$$

  The proof is automatically generated by constructing a term of the corresponding type. It uses a number of intermediate lemmas stating the validity of the considered schedule w.r.t. dependencies.
- We then eliminate time in the preceding theorem, and get for each variable, a lemma stating the desired result.

  We now have proved that any index in the domain of a variable is "accessible".

3.4 Proof irrelevance

To be able to get rid of the accessibility term in the variable definition, we now must prove that the value of a variable instance does not depend on the way its indices are accessible. More precisely, we prove the following property: for any variable X_j and for any H_1 and H_2 of type $(\text{Acc}_{X_j}\ i_1 \ldots i_{n_j})$ we have

$$(X_j\_f\ i_1 \ldots i_{n_j}\ H_1) = (X_j\_f\ i_1 \ldots i_{n_j}\ H_2)$$

Once more, the proof is automatically generated. It makes use of the inductive type of Section 3.2, under the form of an induction principle that is systematically generated by Coq for each inductive type.

---

[1] More precisely, we construct a term of type $(\text{Acc}_{X_j}\ i_1 \ldots i_{n_j})$ under the hypothesis that $(\ i_1 \ldots i_{n_j})$ is in $\mathcal{D}_{X_j}$.

3.5   Undefined values

The properties we have established so far allow us to handle indices that are in the domain of a variable: for these indices, we now are in position to define the corresponding variable value. For indices that are out of the considered domain, we assume the existence of a particular undefined value which will be denoted by $\perp^2$.

The "final" definition of variables is thus given by

$$(\mathtt{X}_j \ i_1 \ldots i_{n_j}) = \mathit{if} \ (i_1, \ldots, i_{n_j}) \ \in \mathcal{D}_{X_j} \ \mathit{then} \ (\mathtt{X}_j\_\mathtt{f} \ i_1 \ldots i_{n_j} \ acc) \ \mathit{else} \ \perp$$

where acc is a "proof" that $(i_1, \ldots, i_{n_j})$ is an accessible index vector knowing that (i) it is in the domain of X_j and (ii) any index in this domain is accessible.

3.6   Fixpoint properties extraction

We finally have to gather all preceding definitions and theorems to state the fact that a given variable is indeed a fixpoint for its defining equation. This results in a theorem that has exactly the same aspect as the initial recurrence equation.

To sum up the whole process, all the user has to do is to write the initial SARE, and provide a schedule (which is in most cases computed by the MMALPHA environment). All theorems mentioned above are then automatically generated and proved.

4   Verification Strategies

Once the system is translated, the user writes down a set of properties he wants to check. During the translation of the system, we create tactics which will facilitate the proofs of these properties and, in the best cases, automate them.

4.1   Using fixpoint properties

When proving a specification for an ALPHA program, the user has to replace variable instances $X[z]$ by their definitions. This is made possible by the previous fixpoint properties theorems, but the use of these theorems is a bit technical: you first have to prove that z is in the domain of X and then, if X is defined with a `case` expression, explore each branch of this expression. We generate three tactics to automate this work.

1. The translation program first defines a tactic which looks in the current assumptions for two affine constraints corresponding to two different branches of a `case` expression. If the search succeeds, a lemma about branches disjointness can end up the current goal.

---

[2] More precisely, we define a particular value for each base type: $\perp$ for boolean values will be named `boolnone`, etc.

2. To prove that an affine constraint is ensured, it may be useful to unfold all domain predicates in the current goal and in the assumptions. The tactic `UnfoldAllDom` makes this work. This is not done before to improve the readability during the interactive proof.

3. Although these two tactics may be called by the user, they are often indirectly called by the third tactic `RewriteX` which, for a given variable X, replaces the first instance $X[z]$ in the current goal by its definition, automatically proves that z is in the domain of X, and rejects all `case` branches which are not compatible with the current assumptions (*in fine*, this instance has been replaced only by the subexpression found in the corresponding branch).

These tactics are written in the tactic langage $\mathcal{L}_{tac}$ of Coq [7]. As constraints are *ground Presburger formulae* [8], solving them reduces to a decidable problem wich is handled by the Coq tactic `Omega`.

4.2 Induction principle associated to a system

Many proofs about SAREs are done by induction, due to the obvious recursive structure of such systems. In the general case, the induction principle for a given SARE relies on a quite intricate well-founded order. We directly make use of the accessibility inductive type of Section 3.2 to generate an induction scheme that exactly maps the structure of the differents proofs we may have to do on the system. When defining an inductive type in Coq, an induction principle is systematically associated to it. In our case, we simultaneously defined an inductive accessibility type for all variables. This definition yields one induction scheme for each variable. We collect together all these schemes in a general induction scheme for the whole system by using the Coq Scheme feature [15]. In this generated induction principle, accessibility predicates appear as arguments just like in the initial mutually recursive variable definition of Section 3.2. We finally eliminate these accessibility arguments to get a more tractable induction scheme. The final induction scheme has the general form displayed in Figure 4. The particular case for the DLMS example of Section 2 is displayed in Figure 5

This scheme associates one property to each polyhedral domain, and can be seen as a "generalized invariant rule": instead of dealing with one invariant for a single loop, we have to manipulate multiple mutually dependent invariants, and the structure of the induction principle reflects the structure of the program's dependencies. In other words, the induction principle captures the "static" part of the semantics (the dependence scheme) while the invariant instantiations reflect the "dynamic" part (value computations between variable instances).

From a practical point of view, all the user has to do is to provide these invariants by instantiating each domain property.

$$(\texttt{P}_{X_1} : (\texttt{Z}^{n_1} \rightarrow \ \texttt{Prop}) \ \ldots \ \texttt{P}_{X_p} : (\texttt{Z}^{n_p} \rightarrow \ \texttt{Prop}))$$
$$((i_1, \ldots, i_{n_1} : \texttt{Z})(\texttt{Dom}_{X_1} \ i_1 \ldots i_{n_1}) \ \rightarrow \ <\mathcal{D}_{X_{1,1}}> \ \rightarrow \ (\texttt{P}_{X_1} \ i_1 \ldots i_{n_1}))$$
$$\ldots$$
$$\rightarrow \ ((i_1, \ldots, i_{n_1} : \texttt{Z})(\texttt{Dom}_{X_1} \ i_1 \ldots i_{n_1}) \ \rightarrow \ <\mathcal{D}_{X_{1,l_1}}> \ \rightarrow$$
$$\cdots \ (\texttt{P}_{X_j} \ (d_{1,l_1,j} \ i_1 \ldots i_{n_1})) \cdots \ \rightarrow \ (\texttt{P}_{X_1} \ i_1 \ldots i_{n_1}))$$
$$\vdots$$
$$\rightarrow \ ((i_1, \ldots, i_{n_p} : \texttt{Z})(\texttt{Dom}_{X_p} \ i_1 \ldots i_{n_p}) \ \rightarrow \ <\mathcal{D}_{X_{p,1}}> \ \rightarrow \ (\texttt{P}_{X_p} \ i_1 \ldots i_{n_p}))$$
$$\ldots$$
$$\rightarrow \ ((i_1, \ldots, i_{n_p} : \texttt{Z})(\texttt{Dom}_{X_p} \ i_1 \ldots i_{n_p}) \ \rightarrow \ <\mathcal{D}_{X_{p,l_p}}> \ \rightarrow$$
$$\cdots \ (\texttt{P}_{X_j} \ (d_{p,l_p,j} \ i_1 \ldots i_{n_p})) \cdots \ \rightarrow (\texttt{P}_{X_p} \ i_1 \ldots i_{n_p}))$$
$$\rightarrow \ ((i_1, \ldots, i_{n_k} : \texttt{Z})(\texttt{Dom}_{X_k} \ i_1 \ldots i_{n_k}) \ \rightarrow \ (\texttt{P}_{X_k} \ i_1 \ldots i_{n_k}))$$

Fig. 4. Generated induction scheme for variable X_k

```
(P_W,P_Y:(Z->Z->Prop); P_E,P_res,P_x,P_d:(Z->Prop))
    ((n,i:Z)(Dom_W n i)->'(-1)+[D]-n+[N] >= 0'->(P_W n i))
    ->((n,i:Z)(Dom_W n i) ->'(-[D])+n-[N] >= 0'->
        (P_W '(-1)+n' i)->(P_E '(-[D])+n')->(P_x '(-[D])-i+n')->
         (P_W n i))
    ->((n,i:Z)(Dom_Y n i)->'1+i = 0'->(P_Y n i))
    ->((n,i:Z)(Dom_Y n i)->'i >= 0'->
        (P_Y n '(-1)+i')->(P_W n i)->(P_x '(-i)+n')->(P_Y n i))
    ->((n:Z)(Dom_E n)->(P_d n)->(P_res n)->(P_E n))
    ->((n:Z)(Dom_res n)->(P_Y n '(-1)+[N]')->(P_res n))
    ->((n:Z)(Dom_x n)->(P_x n))
    ->((n:Z)(Dom_d n)->(P_d n))
    ->(n:Z)(Dom_res n)->(P_res n)
```

Fig. 5. Generated induction scheme for the DLMS filter

```
system Times : {W | 2<=W}
                  (A : {b | 0<=b<=W-1} of boolean;
                   B : {b | 0<=b<=W-1} of boolean)
        returns (X : {b | 0<=b<=W-1} of boolean);
var
  P  : {b,m | 0<=b<=W-1; 0<=m<=W-1} of boolean;
  Si : {b,m | 0<=b<=W-1; 0<=m<=W-1} of boolean;
  So : {b,m | 0<=b<=W; 0<=m<=W-1} of boolean;
  Cin : {b,m | 0<=b<=W-1; 0<=m<=W-1} of boolean;
  XF : {b,m | 0<=b<=W-1; 0<=m<=W-1} of boolean;
  CoutF : {b,m | 0<=b<=W-1; 0<=m<=W-1} of boolean;
let
  P[b,m] = A[b] and B[m];
  Cin[b,m] =
      case
        { | b=0; 0<=m<=W-1} : False[];
        { | 1<=b<=W; 0<=m<=W-1} : CoutF[b-1,m];
      esac;
  XF[b,m] = Si xor P xor Cin;
  CoutF[b,m] = Si and P or Si and Cin or P and Cin;
  So[b,m] =
      case
        { | 0<=b<=W-1; 0<=m<=W-1} : XF;
        { | b=W; 0<=m<=W-1} : CoutF[W-1,m];
      esac;
  Si[b,m] =
      case
        { | m=0} : False[];
        { | 0<=b<=W-1; 1<=m<=W} : So[b+1,m-1];
      esac;
  X[b] = So[b+1,W-1];
tel;
```

Fig. 6. ALPHA system computing binary product

5 Example

5.1 Binary product

The program displayed in Figure 6 computes the binary product of two arrays
A and B of W bits. Output variable X is an array of W bits corresponding to
the W most significant bits of the product. We proved with Coq this functional
specification. We detail here the successive verification steps.

1. First of all, we developed a Coq module handling conversions between bit
   vectors (represented by functions of type Z->bool) and integers. The main
   functions of this module are

   $$\texttt{bool2Z : bool -> Z}$$

which converts one bit into an integer, and

$$\texttt{bin2Z : (Z->bool) -> Z -> Z}$$

which, given a function $\texttt{f}$ of type $\texttt{Z->bool}$ and a positive integer k, computes the integer n represented by the $k+1$ bits $(\texttt{f } 0), \dots, (\texttt{f } k)$: $n = \sum_{i=0}^{k} 2^i \cdot (\texttt{f } i)$.

2. The system is loaded in MMALPHA. The following schedule is proposed:

$$\begin{aligned}
t_X[b] &= 1 + 2b + 5W & t_P[b,m] &= 1 \\
t_{Si}[b,m] &= 1 + 2b + 5m & t_{So}[b,m] &= 3 + 2b + 5m \\
t_{Cin}[b,m] &= 1 + 2b + 5m & t_{XF}[b,m] &= 2 + 2b + 5m \\
t_{CoutF}[b,m] &= 2 + 2b + 5m
\end{aligned}$$

3. The translation function is called on the system and the corresponding Coq file `Times.v` is generated.

4. At the end of the generated Coq file we add the following theorem:

```
Theorem X_mult :
  '(bin2Z X (W-1))=((bin2Z A (W-1))*(bin2Z B (W-1)))/(two_p W)'.
```

where $(\texttt{two\_p } W)$ represents 2^W.

5. We introduce a number of intermediate lemmas about local variables. Some of these lemmas deal with non trivial mathematical results on euclidean division. The generated tactics allow us to focus only on the "conceptual" parts of the proof. In particular the tactics handling variable rewritings and case branches exploration save a considerable amount of tedious work. In this particular example, we didn't make use of the induction scheme, since the involved dependencies are quite simple, and most of the work is dedicated to conversions from bit vectors to integers.

The corresponding Coq file is available at [1].

5.2 DLMS

To illustrate the use of the induction principle associated to an ALPHA system, we show here how we prove a simple property of the DLMS system which requires multiple mutual induction. The theorem we want to prove is the following

Theorem 2. $(\forall n \in \mathcal{D}_\texttt{d}, \ (\texttt{d } n) = 0) \Rightarrow (\forall n \in \mathcal{D}_\texttt{res}, \ (\texttt{res } n) = 0)$

According to the dependencies of the DLMS system, the proof of this theorem requires an additional lemma expressing the fact that "all instances of $\texttt{Y}$ are equal to zero". But this lemma requires "all instances of $\texttt{X}$ are equal to zero" to be proved, which in turn depends on "all instance of $\texttt{E}$ are equal to zero". This last property itself depends on the nullity of the instances of $\texttt{res}$. This cycle in the proof process shows the technical difficulties of reasoning about mutual recursive functions.

Thanks to the induction principle generated and proved during the translation process, all these lemmas and the final theorem can be proved together. We just have to apply the induction principle with the relevant properties. This results in the following Coq proof command:

```
Apply res_induction
        with  P_W:=[n,i:Z](W n i)='0'  P_Y:=[n,i:Z](Y n i)='0'
              P_E:=[n:Z](E n)='0'      P_res:=[n:Z](res n)='0'
              P_d:=[n:Z](d n)='0'      P_x:=[n:Z]True.
```

We now have 9 subgoals to prove, the majority of them are obvious and are automatically discharged by Coq. The theorem is thus finally proved with a few 10 lines of proof commands. After closing the Coq section, the final exported version of the theorem is:

```
(Znone,N,M,D:Z; x,d:(Z->Z);
  paramsCond:('(-1)-D+M-N >= 0'/\'(-1)+D-N >= 0'/\'(-3)+N >= 0'))
  ((n:Z)([n:Z]'M-n >= 0'/\'n-N >= 0' n)->'(d n) = 0') ->
   (n:Z)([n:Z]'M-n >= 0'/\'n-N >= 0' n)
            ->'(res Znone N M D x d paramsCond n) = 0'
```

This form illustrates the assumptions we made during the Coq section, *i.e.*, the output variable res depends on the two input variables x and d, but also on the undefined value Znone, the 3 parameters N, M, D and on a proof (paramsCond) that these parameters satisfy the system constraints.

6   Related Work

When using theorem proving tools to model and verify systems or programs, one has to chose between formalizing the program's semantics or directly translating its syntactic constructs into "equivalent" ones. In the case of an imperative language, we would have to model its constructs under the form of denotational, operational or axiomatic semantics (see [13] for an example of a translation of a simple imperative language in PVS and further references, or [9] for a translation of the semantics of an imperative language in Coq). Translation of an equational language into Coq has been done with the reactive language Signal [12]. In this work, the co-inductive trace semantics of Signal is implemented as a set of Coq libraries that model the primitives of the language and provide basic lemmas and theorems. The translation of a particular system is done "by hand" using these primitives, and equations are translated as axioms.

Modelling mutual recursive functions in type theory is a difficult problem, since one has to prove function termination by using only structural recursive definitions. We may either try to prove termination via a structural ordering on value domains (see [2] for instance for this kind of approach and related work), or add to the initial function definition an argument which is known to be structurally recursive. In [5], a general method is given for the latter approach: a first definition of functions is given, which includes in its arguments a special-purpose accessibility predicate. This predicate is then proved true for all inputs. This allows for giving a simpler version of the function without accessibility predicates. In our approach we take advantage of the possible existence of a scheduling to (i) automatically generate the proof that all inputs (in our case, domain indices)

are accessible and (ii) automatically generate a theorem stating that the variables are indeed a fixpoint for the initial equations. In [4], an approach based on the HOL theorem prover is used to construct induction schemes for mutually recursive functions. They develop specific algorithms for this construction. Due to the use of Coq and of its type-theoretic nature, we cannot use the same techniques. When dealing with SAREs, the use of measures would have given an easier definition of recursive functions in PVS and HOL/Isabelle. Nevertheless, most of the theorems we have to prove here would also have been necessary with this other technique. Anyway, the additional technical lemmas are fully automatically discharged.

Though many properties concerning parameterized SAREs are undecidable, recent work gives heuristics for automatically checking equivalence of two SAREs in some particular cases [3]. By using a general-purpose theorem prover, we considerably extend the possibilities of verification to complex properties. This work extends a previous similar one that had been implemented with PVS [6]. In this previous work however, induction schemes were not generated but had to be explicitly provided by the user. Moreover fixpoint equations were defined as axioms.

7   Conclusion

We have presented a tool and a method for proving properties on parameterized systems of affine recurrence equations. We proceed by translating SAREs into the specification language of the Coq theorem prover. We make an extensive use of the specific features of the polyhedral model to automatically generate Coq theorems that will be usable in a further proof process. For a given system, we take advantage of the possible existence of a particular schedule to get a functional translation of recurrence equations where accessibility predicates have been eliminated. The translation relies on the definition of a mutual inductive type, that will be used at the same time to get a functional version of recurrence equations and an induction scheme that is specific to the translated system.

We partially overcome the lack of automation by generating tactics adapted to each system's structure. These tactics implement complex induction schemes or theorem instantiations. For a given system, this allows to factorize and automate the most technical steps of the proofs.

Much work remains to be done. First, we need to extend this work to hierarchically structured systems, to be able to handle more complex systems and reuse proofs in a modular manner. We could also combine the different tactics we developed in a more general strategy that would explore a bounded-depth tree of rewritings, use of previously proved theorems and generated tactics. We also could study the specialization of the induction scheme to the problem of deciding wether two SAREs are equivalent, in order to get an automatic equivalence checker. It would finally be valuable to take advantage of a better interaction between Coq and the static polyhedral analysis performed by the MMAlpha tool, to decrease the number of calls made to the Omega library.

References

1. `http://www.irisa.fr/lande/pichardie/coqalpha/`.
2. A. Abel. Specification and verification of a formal system for structurally recursive functions. In *Types for Proof and Programs, International Workshop, TYPES '99*, volume 1956 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2000.
3. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. Technical Report 4285, INRIA, 2001.
4. R. J. Boulton and K. Slind. Automatic derivation and application of induction schemes for mutually recursive functions. In *First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 629–643, London, UK, July 2000. Springer-Verlag.
5. A. Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Sweden, nov 2002.
6. D. Cachera, P. Quinton, S. Rajopadhye, and T. Risset. Proving properties of multidimensional recurrences with application to regular parallel algorithms. In *FMPPTA'01*, San Francisco, CA, April 2001.
7. D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*, volume 1955, pages 85–95, Reunion Island, November 2000. Springer-Verlag LNCS/LNAI.
8. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal of Computing*, 1975.
9. J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.
10. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.
11. M. Katsushige, N. Kiyoshi, and K. Hitoshi. Pipelined LMS Adaptative Filter Using a New Look-Ahead Transformation. *IEEE Transactions on Circuits and Systems*, 46:51–55, January 1999.
12. D. Nowak, J.R. Beauvais, and J.P. Talpin. Co-inductive axiomatization of a synchronous language. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 387–399. Springer-Verlag, September 1998.
13. H. Pfeifer, A. Dold, F. W. von Henke, and H. Ruess. Mechanised semantics of simple imperative programming constructs. Technical Report UIB-96-11, Ulm University, dec 1996.
14. Y. Saouter and P. Quinton. Computability of Recurrence Equations. Technical Report 1203, IRISA, 1990.
15. LogiCal Project The Coq Development Team. *The Coq proof Assistant, Reference Manual*.
16. D. Wilde. A library for doing polyhedral operations. Technical Report 785, Irisa, Rennes, France, 1993.
17. D. K. Wilde. The Alpha language. Technical Report 999, IRISA, Rennes, France, January 1994.