Abstract

We present a set of verification methods to prove properties of parallel systems described by means of multidimensional affine recurrence equations. We use polyhedral analysis and transformation techniques together with theorem proving. Polyhedral techniques allow us to handle simple but otherwise costly proof steps, while theorem proving provides more expressivity and more complex proof techniques. This allows large, generic and structured systems to be verified. These methods are implemented in the MMAlpha environment using the PVS theorem prover.

1 Introduction

Systems of recurrence equations [9] provide a useful foundation for designing parallel multidimensional arrays for computation intensive applications. Many authors have investigated the so called space-time mappings or index transformations to obtain dedicated VLSI processor arrays from an initial system of recurrence equations. Moreover, the tools and synthesis methods developed in this context have a strong relationship with loop parallelization methods.

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 [18] and the MMALPHA environment [17] provide the syntax and tools to handle such recurrence equations. The MMALPHA environment implements polyhedral transformations that allow us to start from a high level specification and refine it to a synthesizable architectural description. It is also possible to derive 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. Furthermore certain properties, if expressed in the SARE formalism, can be verified directly within the polyhedral model by MMALPHA. Nevertheless, there are many cases when we need to formally prove certain properties that are not ensured by the refinement process. This is the case for instance, when a refinement step does not preserve the semantics of a set of equations, or when we want to prove a property that is not expressible in the formalism of affine recurrence equations. More generally, the development of more complex and structured ALPHA programs [7] has called for the elaboration of verification methodologies.

We present here a set of proof methods that can be used to formally verify properties of SAREs. Many embeddings of languages in logical frameworks like HOL, Coq, PVS, etc. have been investigated (see [12, 2, 5] for instance). The originality of our approach is that it combines semantics-preserving polyhedral transformations with calls to a theorem prover. On one hand, the polyhedral model provides powerful tools (static analysis on polyhedral bounds, rewriting rules, verification of certain inductive properties) for handling “simple” proof steps which would be extremely inefficient in a general purpose prover. On the other hand, theorem proving provides the additional expressivity and proof mechanisms that allow us to perform more complex proofs. For those properties that need a tool more powerful than the polyhedral model, we have chosen to use a theorem proving environment based on higher order logic, namely PVS. Our main motivations are (i) the expressivity of higher order logic; (ii) the ability to handle generic descriptions with unspecified size parameters (as ALPHA systems are); (iii) the possibility to easily construct structured proofs and (iv) the ability to use induction schemes.

This paper is organized as follows. In section 2, we briefly present the ALPHA language and the MMALPHA environment. In section 3, we present the PVS theorem prover. Section 4 describes our methodologies, section 5 gives examples, and we conclude in section 6.

2 The Alpha language

The ALPHA language was initially designed by Mauras [10] for the synthesis of regular architectures. It is based on the model of recurrence equations, introduced by Karp, Miller and Winograd [9]. In this section, we briefly introduce the principles and notations of the ALPHA language. See [18] for more details. In the following, we denote by \( \mathbb{Z} \) and \( \mathbb{Q} \) the sets of integers and rational numbers,
2.1 Syntax

2.1.1 Domains and variables

All data structures manipulated in ALPHA are defined over polyhedral domains and may be viewed as “polyhedral shaped” arrays. A polyhedron of dimension \( n \) is a subspace of \( \mathbb{Q}^n \) bounded by a finite number of hyperplanes. A polyhedral domain of dimension \( n \) is defined as the intersection of a polyhedron of dimension \( n \) with \( \mathbb{Z}^n \).

A variable is a mapping from a domain (of its indices) to values in a basic data type (integer, boolean or real). Scalar variables are defined over the trivial domain \( \mathbb{Z}^0 \).

2.1.2 Programs

ALPHA is a strongly typed, structured equational language. A system (program) in ALPHA consists of the declaration of variables, parameters, declarations of (i) input, (ii) output, (iii) local variables, and a set of affine recurrence equations, delimited by the keywords let and tel, recursively defining output and local variables.

As an example, we give a program that implements a convolution filter, taken from [18]. The output of this program is computed as \( y[i] = \sum_{j=1}^{N} a[j] * x[i-j+1] \). The corresponding ALPHA program is displayed in figure 1.

```
system convolution { N | 1<=N }
    { a ; { i | 1<=i<=N } of integer; } returns ( y : { i | i>=1 } of integer; )
    var
    Y : { i,j | 0<=j<=N ; i>=N } of integer;
    let
    Y[i,j] = case
        { j=0 } : 0;
        { i>=j=N } : Y[i,j-1] + a[j] * x[i-j+1]; esac;
    y[i] = Y[i,N];
    tel;
```

**Figure 1. Alpha program computing a convolution filter**

In this example, expressions such as \( \{ j | 1<=j<=N \} \) denote domains, \( N \) is a parameter (the width of the convolution window), \( X \) and \( a \) are the input variables, \( Y \) is the output variable and \( Y \) is a local variable. Note that a program may have several output variables.

Due to the equational nature of the language, ALPHA programs respect the substitution principle: any instance of a variable in any expression may be replaced by the right hand side of the equation defining that variable. Moreover, ALPHA is a single assignment language, and the equations in a system are unordered.

2.1.3 Operators

The expressions appearing on the right hand side of equations are obtained by combining variables or constants by means of two kinds of operators: pointwise operators and spatial operators.  

Pointwise operators are the classical data-parallel componentwise generalizations of scalar operators. The domain of an expression defined by means of a pointwise operator is defined to be the intersection of the domains of the subexpressions combined by that operator.

Spatial operators are used to manipulate domains. ALPHA has three spatial operators, namely the dependence, restriction and case operators.

- The dependence operator. A dependence function is a mapping from \( \mathbb{Z}^n \) to \( \mathbb{Z}^m \), where \( n, m \in \mathbb{N} \). It is denoted by \( (i, j, \ldots \rightarrow f(i, j, \ldots)) \), where \( f \) is an affine function. The dependence operator “\( /\)" combines an expression and a dependence function. For instance, \( X.(i, j \rightarrow j, i) \) is the transpose of a two-dimensional variable \( X \). The domain of this expression is the preimage of the domain of \( X \) by \( f \).

- The restriction operator. It restricts the domain of an expression by means of affine constraints. For instance, \( \{i, j \ | \ i \geq 0\} : X \) restricts mapping \( X \) to the intersection of the domain of \( X \) with the half-plane \( i \geq 0 \). Its domain is the intersection of this half-space with the domain of \( X \).

- The case operator. It pieces together a set of disjoint subexpressions. The value of expression case \( E_1; \ldots; E_m \) at index \( z \) is the value of expression \( E_k \) if \( z \) is in the domain of \( E_k \). To be valid, the branches of the case must have disjoint domains. The resulting domain is the union of the domains of all subexpressions.

2.2 Semantics

Due to the equational nature of ALPHA and the single assignment property, its semantics were first given denotational semantics [10]. We will not go into details here. Let us simply recall that the denotational semantics of an ALPHA system are computed as the least fixpoint of the system of equations it defines. Each instance of a variable possibly evolves from undefined into valued, and does not change once it has been given a value.

Operational semantics have been defined as a preamble to the definition of invariant-based proof methodologies [3]. Practically interesting ALPHA programs belong to a class called schedulable systems, i.e., systems for which we are able to define an execution order on computations (a timing function) compatible with the dependence relations. If we restrict ourselves to schedulable systems, denotational and operational semantics are equivalent. Nevertheless, testing whether a system is schedulable is known to be undecidable [15].

---

1In addition, there is a reduction operator which is not described here.
2.3 The MMAlpha environment

MMALPHA is an environment that implements a number of manipulations on ALPHA programs. It is written in C and Mathematica, but the user only sees the Mathematica interface. MMALPHA uses the symbolic computation facilities of Mathematica together with powerful transformations on polyhedra written in C within the polyhedral library [17].

The basic principle of the MMALPHA environment is the following: Mathematica stores an internal representation of an ALPHA program as an abstract syntax tree and performs computations on this internal representation via user’s commands or functions. The environment provides a set of predefined commands and functions to achieve the following purposes.

- Static analysis of programs, including analysis of polyhedral domains’ shapes and sizes.
- Simulation, execution through standard interpretation and a memoizing optimization.
- Architectural description and VHDL generation.
- Transformations of ALPHA programs. These transformations are based on polyhedra manipulations and include pipelining of variables, scheduling, change of basis, etc. One of these transformations, called normalization, will prove very useful for our purposes. For any ALPHA program, this transformation computes its case-restriction-dependency normal form, i.e., an equivalent program where each variable is defined by means of an unique case expression, and all dependence functions are shifted at the lowest level in the AST (dependence functions are directly applied to variables).

3 The theorem proving approach

Theorem proving is the paradigm of mechanically deriving and verifying proofs of mathematical assertions. These assertions have to be expressed and manipulated in some predicate logic. Existing theorem provers can be classified into two categories, depending on the nature of this logic. The first class contains the systems based on first order logic, and the second one relies on some kind of higher order logic, allowing greater expressibility and more powerful proof mechanisms. An example in the latter class is the theorem-proving assistant HOL, based on natural deduction [8]. This system has been widely used in the domain of hardware verification. Specification languages based on higher order logic allow one to define and manipulate usual mathematical objects such as sets, functions, logical formulas, or even proofs in systems like Coq [1]. The specification language is supposed to be powerful enough to describe systems and express their properties. The prover itself usually consists of an interactive environment, that constructs proofs under user guidance. From the user’s point of view, a proof consists of a sequence of elementary proof steps or applications of proof strategies (i.e., combinations of elementary steps).

The PVS system is developed at SRI. It consists mainly of a specification language [11] and a theorem prover [16]. As explained below, the specification language is used to write theories, that are checked by the prover.

The PVS Language

A PVS specification consists of a collection of theories. Each theory may contain assumptions, function or predicate definitions, axioms, and theorems. The PVS base library (called prelude) includes the definition of basic mathematical objects, together with their properties. Additional specific libraries are provided. Writing a specification thus consists in writing one or more new theories in the specification language. The specification language of PVS is based on classical, typed higher-order logic. The base types include uninterpreted types, including user-defined types, and built-in types such as the booleans, integers, reals and some ordinals. The type-constructors include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types, such as lists and binary trees. Predicate subtypes and dependent types can be used to introduce constraints (for example we can define a prime-number type using such constraints). A theory can be parametric in certain specified types and values. Assumptions are thus constraints on the parameters of a theory. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, within a natural syntax. Functions can either be fully specified, or only be given a type, their behaviour being described by a set of axioms. Axioms, theorems and lemmas belong to the same class of formulas.

The PVS Prover

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. They include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. The implementations of these primitive inferences are optimised for large proofs: for example, propositional simplification uses BDDs, and auto-rewrites are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies. Such strategies, if correctly adapted to the properties they are supposed to prove, may yield automatic proofs. Proofs yield scripts that can be edited, attached to additional formulas, and rerun.

4 Checking for functional properties

The theorem proving approach is particularly well-suited to proving functional properties, i.e., predicates that relate
output values to input values. Of course, the validity of a predicate depends on a particular semantics. As mentioned in section 2.2, we focus here on the denotational semantics.

We have developed two complementary proof methodologies. The choice between them mainly depends on the nature of the properties the user wants to establish and results in two different ways to use the prover. The first method is well-suited for simple properties. It asks PVS to check tautologies expressed within the ALPHA language. This method is ideally suited for mechanization, as we shall see later. The second one may be used for more complex properties and structured systems, and makes a more extensive use of the PVS specification language and theorem proving features.

Depending on the nature of the property we want to prove, we have to chose between the methods proposed here. Of course, simple properties can be proved by uniquely calling PVS. Nevertheless, better performance is obtained by applying first as many MMALPHA transformations as possible, to benefit from the powerful polyhedral manipulations.

4.1 Reducing to tautologies

This first approach is used when we want to check properties that are expressible in the ALPHA language. The general method consists of expressing the desired property as a boolean ALPHA expression and then checking that this expression is “trivially” true over its entire domain. By “trivially” true we mean that we only want to check arithmetic and logic properties. The proof is thus split into two parts, handled by distinct tools. The first part, handled by MMALPHA, consists of “constructing” the tautologies by using syntactic rewriting rules. The second part consists of checking these tautologies with PVS. There are two cases, depending on wether the proof of the property involves recursion or not, and resulting in two different ways to use MMALPHA. Note that in this first approach, the inductive part of the proof, if any, is not handled by PVS but within MMALPHA itself.

4.1.1 The simplest case

This case arises when no induction at all is to be used. After having substituted some variables whose names appear in the expression of the property by their defining expressions, we just have to check a set of trivial properties. The proof process is the following.

- substitute some variables by their defining expressions using the MMALPHA substituteInDef command;
- normalize this new system;
- “send” the normalized expression to PVS, by means of the pvsTautoCheck command.

A PVS window pops up and the PVS proof checker tries to establish all the tautologies corresponding to each branch of the normalised case expression. The proof status is then summarized.

This method works for very simple properties, for instance if we want to prove that a slight change in a computation does not affect the semantics of the entire system (e.g. adding a new intermediate variable or changing the way a variable is computed without changing the underlying algorithm). We also can prove less trivial properties, for instance, checking the validity of a proposed schedule for a system of recurrence equations. Complete examples are given in section 5.

4.1.2 Handling recursion within Alpha

Most of the time, even if we are able to express the desired property within the ALPHA language, it remains impossible to derive tautologies that are simple enough: since values are recursively defined on domains whose size depends on arbitrary parameters—and may even be infinite—we generally cannot know in advance the number of syntactic substitutions that would yield an expression free of dependences.

So, in such cases, a proof by induction has to be performed. We now show how this proof can be performed in a simple manner within the MMALPHA environment. Let us explain the method on the general base case of such a proof. Let us assume that we want to prove functional equivalence of two systems. Let us also assume that the first system uses a general recursive scheme to compute its output, while the second one computes the same result, but without involving recursion, i.e., its result depends only on index values or input variables (it is a closed form solution). Let us finally assume that the first system admits a valid schedule. Proving the equivalence between these two systems is done in three steps, involving three distinct “tools”.

1. express with an ALPHA boolean expression the fact that the values of both systems follow the same recursion scheme; this is a purely syntactic step;

2. prove with PVS that this fact is a tautology;

3. use a “meta” argument to conclude: under the schedulability assumption, we can conclude that the two systems are equivalent.

Let us detail these three steps. For this purpose, let us call Rec the output of the first system and Closed that of the second one.

1. We apply the same technique as in the case of trivial tautologies, but with a few more steps.

   - begin by defining a variable the RHS of whose equation is the expression Rec = Closed;

   - substitute some variables by their defining expressions using the MMALPHA substituteInDef command;
   - normalize this new system;
   - “send” the normalized expression to PVS, by means of the pvsTautoCheck command.

   A PVS window pops up and the PVS proof checker tries to establish all the tautologies corresponding to each branch of the normalised case expression. The proof status is then summarized.

   This method works for very simple properties, for instance if we want to prove that a slight change in a computation does not affect the semantics of the entire system (e.g. adding a new intermediate variable or changing the way a variable is computed without changing the underlying algorithm). We also can prove less trivial properties, for instance, checking the validity of a proposed schedule for a system of recurrence equations. Complete examples are given in section 5.

   4.1.2 Handling recursion within Alpha

   Most of the time, even if we are able to express the desired property within the ALPHA language, it remains impossible to derive tautologies that are simple enough: since values are recursively defined on domains whose size depends on arbitrary parameters—and may even be infinite—we generally cannot know in advance the number of syntactic substitutions that would yield an expression free of dependences.

   So, in such cases, a proof by induction has to be performed. We now show how this proof can be performed in a simple manner within the MMALPHA environment. Let us explain the method on the general base case of such a proof. Let us assume that we want to prove functional equivalence of two systems. Let us also assume that the first system uses a general recursive scheme to compute its output, while the second one computes the same result, but without involving recursion, i.e., its result depends only on index values or input variables (it is a closed form solution). Let us finally assume that the first system admits a valid schedule. Proving the equivalence between these two systems is done in three steps, involving three distinct “tools”.

   1. express with an ALPHA boolean expression the fact that the values of both systems follow the same recursion scheme; this is a purely syntactic step;

   2. prove with PVS that this fact is a tautology;

   3. use a “meta” argument to conclude: under the schedulability assumption, we can conclude that the two systems are equivalent.

   Let us detail these three steps. For this purpose, let us call Rec the output of the first system and Closed that of the second one.

   1. We apply the same technique as in the case of trivial tautologies, but with a few more steps.

   - begin by defining a variable the RHS of whose equation is the expression Rec = Closed;

   - substitute some variables by their defining expressions using the MMALPHA substituteInDef command;
   - normalize this new system;
   - “send” the normalized expression to PVS, by means of the pvsTautoCheck command.

   A PVS window pops up and the PVS proof checker tries to establish all the tautologies corresponding to each branch of the normalised case expression. The proof status is then summarized.

   This method works for very simple properties, for instance if we want to prove that a slight change in a computation does not affect the semantics of the entire system (e.g. adding a new intermediate variable or changing the way a variable is computed without changing the underlying algorithm). We also can prove less trivial properties, for instance, checking the validity of a proposed schedule for a system of recurrence equations. Complete examples are given in section 5.
• substitute \(Rec\) by its defining expression (this expression has \(Rec\) as a free variable);
• substitute all occurrences of the variable \(Rec\) in the resulting expression by the variable \(Closed\).

At this stage, we just have expressed the fact that, if \(Rec\) is defined by the fixpoint equation \(Rec = F(Rec)\), then \(Closed\) is a solution for the same equation: \(Closed = F(Closed)\).

• substitute all occurrences of \(Closed\) by their defining expression;
• normalize.

At this stage, as the defining expression of \(Closed\) does not contain any occurrence of \(Closed\), we have a set of trivial tautologies.

2. send this set of tautologies to PVS; this step is exactly the same as in case 4.1.1;

3. we now have to conclude that we have really proved the equivalence. Intuitively, three arguments are necessary:

• “initial” values, i.e., values that only depend on input or index values, are the same in both systems; this is simply a part of the set of tautologies;
• starting from these initial values, we always generate identical values for both systems; this comes from the fact that the recursion scheme is the same;
• all values are really computed; this comes from the assumption about the existence of a valid schedule.

A formal proof of these facts is given in [4]. Note that this proof is done once and for all, as its only assumption is that the first system admits a valid schedule. As a consequence, once the variables on which recursion has to be performed have been identified, the rest of the process is easily mechanizable.

Of course, this method is also applicable to mutual recursion, as soon as we are able to give a closed form for each variable involved in the considered mutual recursion scheme.

4.2 More expressivity

When possible, expressing properties within the ALPHA language and simply checking tautologies is a simple and intuitive way to check these properties. But if it were only for checking such properties, using a simpler ad-hoc and specific tool in place of PVS would be more suitable. In fact, the use of PVS is justified and motivated by the fact that we want to express more complex properties and perform more complicated proofs.

Specifically, lack of expressivity comes from two kinds of restrictions.

• Using the ALPHA language restricts, de facto, the form of the expressions we are allowed to use. Although all usual arithmetic operators are allowed in ALPHA expressions, a major restriction comes from the fact that only affine index expressions are allowed. For example, we are able to express a non affine expression for a timing function, but we would not be able to use this index expression as an index to denote an instance of a variable in any property.

• When writing an ALPHA expression to denote a property, explicit universal quantification is assumed. This means that the only properties we are able to express are of the form “for any index \(z\) in domain \(D\), property \(P(z)\) holds”, where \(D\) is a polyhedral domain. This does not allow for more complex properties using existential or universal quantification on domains other than unions of convex polyhedra, or quantification on any other object (i.e., higher order logic).

To overcome these restrictions, we allow the user to express properties using all the features of the PVS specification language.

The use of PVS also extends here to a more complex use of the prover. The methodology presented in the previous section, consisting of textual substitutions combined with a semantic argument establishing an inductive property, might not be practicable in all cases. For more complex induction proofs, we directly use the induction schemes of PVS, as detailed below.

We have therefore developed an interface that translates a combination of ALPHA systems and sets of properties into PVS theories and runs PVS according to specified strategies. One of its main interest, in addition to expressing and proving more complex assertions about systems, is the development of structured and hierarchical proofs, i.e., proofs using properties of subcomponents. We detail here the main features of this tool.

Classification of properties Before the translation step proper, the user first has to express the properties to assert or prove. One can assert properties about a given set of variables or subsystems. For instance, input variables may be characterised by some properties about their values. More generally, one can describe the behaviour of local variables or subsystems. In the proof process, it is not necessary to prove these properties as a preliminary result. This proof can be delayed until later. In order to help the tool in generating proof strategies, properties are attached to variables and classified into three distinct categories.

• axioms are properties attached to input variables; they do not need to be proved, but are considered as assumptions about input values.

• simple properties are attached to variables that are not defined recursively or to subsystems; these are properties that do not need PVS induction to be proved; they might also be attached to recursively defined variables,
but in that case we won’t be able to use this definition in the proof.

- inductive properties are attached to recursively defined variables and have to be proved by means of an induction scheme.

Variable abstraction When developing and proving a system in a structured way, one might want to keep some parts at an abstract level, without specifying its implementation. To ensure this possibility, we distinguish between abstract and concrete variables.

- abstract variables are simply translated into the PVS theory as mappings from domains to values, without specifying a value for this mapping. domains themselves are abstracted, i.e., we only know their dimension and not their precise shape. Behaviour of abstract variables is specified by axioms.

- concrete variables are fully translated into the PVS theory; however, their behaviour (i.e., their defining expression) is not translated as a function (a fully described mapping from domains to values), but as a PVS axiom, i.e., a relation between the variable and its defining expression. Using an axiom rather than a function has two advantages: it avoids infinite loops of rewritings (these functions are recursive, as each one represents a recurrence equation), and it allows using mutual recursion, which is not directly allowed in PVS.

Recall that in PVS, axioms and theorems share the same status. A theory remains incomplete unless all axioms it uses have been proved. The axioms describing concrete variables cannot be proved, since they model the program’s behaviour. Of course, starting from an contradictory program may introduce paradoxes. Nevertheless, a program is contradictory if it contains dependence cycles, i.e., if it is not schedulable, and we do not consider this case in practice.

Proof strategies Together with the PVS theory containing variable descriptions and properties, the interface writes a PVS “strategies-file” containing a proof strategy for each property. Two groups of strategies are defined.

- Simple strategies are used for “simple” properties. Examining the defining expressions of variables, they try to determine which properties or axioms are necessary to establish the desired result, using PVS decision procedures and rewriting rules. Moreover, these strategies have to prevent the prover from entering an infinite rewriting loop. This is ensured by disabling automatic rewriting of recursively defined variables.

- Inductive strategies are used for “inductive” properties. In addition to simple strategies, they use the predefined induction schemes of PVS that decompose the proof into a base case and an induction step. The user has to indicate on which expression induction has to be performed (most of the time, it is on a timing function, automatically generated, or “guessed” by the user and proved correct, as in the APP example below).

Modular development and refinement As mentioned above, using a theorem-proving approach is particularly interesting in a modular and hierarchical development process. After having proved—or simply assumed—a property on a subsystem, this property can be used in a proof for a more complex system. Similarly, a proof on a variable or subsystem’s property can be delayed further in a refinement process. We just have to consider this variable as “abstract” until its full specification is given.

5 Application examples

The first two examples use an algorithm for solving the Algebraic Path Problem [14]. This general problem unifies through a single generic algorithm a number of specific problems, like shortest path between all pairs of vertices in a graph, transitive closure of a binary relation, etc. The original APP algorithm is defined on a semi-ring $S(\oplus, \odot)$ with a closure operator, and is shown in figure 2.

```
t[i,j,k] = case
  { | k=0: a[i,j]; --input
    | k>0: i=k; j=k; (i<k ? j<k : j=k)
    | app[i,j,k] semiring | app[i,j,k-1];
  [ | k>0; i=k; (j<k ? j<k : j=k)
    | app[i,j-1,k] semiring | app[i,j,k];
    [ | k>0; (i<k ? j<k : j=k)
    | app[i,j,k] semiring | app[i,j,k-1];
esac;
```

Figure 2. Original APP recurrence, on domain $\{i, j, k \mid 0 \leq k \leq N; 0 < i, j < N\}$

```
app[i,j,k] = case
  [ | k=0: 0[]
    | k>0: i=j=k: t[i,j,k] > max(t[i+1,j+1,k-1],t[0,0,k]);
  [ | k>0; i=k; j<k: t[i,j,k] > max(t[i+1,0,k-1],t[i,0,k]);
    [ | k>0; j=k: t[i,j,k] > max(t[i,0,k-1],t[0,0,k]);
    [ | k>0; i=j=k: 0[];
  [ | k=0: a[i,j]; --input
    [ | k>0; i=k; j=k: (i<k ? j<k : j=k)
    [ | k>0; (i<k ? j<k : j=k)
    [ | k>0; (i<k ? j<k : j=k)
    [ | k>0; (i<k ? j<k : j=k)
    [ | k>0; (i<k ? j<k : j=k)
        [ | k>0; i=k; (j<k ? j<k : j=k)
        [ | k>0; (i<k ? j<k : j=k)
        [ | k>0; (i<k ? j<k : j=k)
        [ | k=0: a[i,j]; --input
        [ | k>0; i=k; j=k: t[i,j,k] > max(t[i+1,j+1,k-1],t[0,0,k]);
        [ | k>0; i=k; j<k: t[i,j,k] > max(t[i+1,0,k-1],t[i,0,k]);
        [ | k>0; j=k: t[i,j,k] > max(t[i,0,k-1],t[0,0,k]);
        [ | k=0: a[i,j]; --input
    esac;
```

Figure 3. Checking a non linear timing function
A non linear schedule  A large variety of algorithms can be derived from the initial system, e.g. by pipelining variables or modifying the domain’s shape [6, 13]. For one of these versions, a particular timing function has been discovered. As it has not been given by an automatic scheduling algorithm, we want to check that this function, namely \( t[i, j, k, N] = (N + 2) \cdot (k - 1) + j \cdot N + i + 1 \), is valid, i.e., that it respects all the dependencies. We just have to write the system displayed on figure 3 and to check that the expression defining \( X \) is a tautology, as explained in section 4.1.1. This is done by writing three MMALPHA commands: substitution, normalization, and call to PVS via a “tautology check” command. After normalization, 40 tautologies are automatically generated and proved.

Free schedule  Another problem we are interested in is that of computing a free schedule for this algorithm, i.e., a schedule that ensures computation of any instance of the app variable as soon as possible. Here, we are not interested in the result of the system, but only in the dependencies contained in this algorithm. For this reason, we only retain this information, forgetting all other aspects of the algorithm. This results in an abstract version of the initial APP recurrence equations, where any expression, say, \( A * B \), has been replaced by \( 1 + \max(A, B) \), thus expressing temporal dependencies. This abstract version is displayed on figure 4, where free is the variable representing the “free” schedule. The domain of the recurrence is the same as before.

We then guess that this free schedule has a closed form depending only on \( k \), but we would like to check it formally.

We thus write a system containing the (recursive) definition of free and the expression of its supposed closed form, together with a boolean variable expressing the equality between these two forms. This finally results in the system of figure 5.

The proof exactly proceeds as mentioned in section 4.1.2 and results in checking 34 trivial tautologies. This is a mechanization of a manual proof first given by Rajopadhye [14].

An example of a structured proof  We finally give a short example of how a structured proof can be constructed. Our example deals with one of the simplest structured ALPHA programs, namely a system computing a matrix product, using subsystems, each computing a matrix-vector product. These systems are separately written into two files, as shown in figure 6.

In this example, we want to prove that the product of two positive matrices is a positive matrix. We would like to use in our proof a lemma that a similar property holds for a matrix-vector product. The associated properties are shown in figure 7.

In these sets of properties, the keyword axiom is used for preconditions, i.e., properties dealing with input variables. Keywords simpleprop and inductprop are used for “simple” and “inductive” properties, respectively, as explained in section 4. When proving the properties attached to a system, axioms are taken as postulates, whereas when using a subsystem, its axioms must be a logical consequence of the caller’s axioms. Note that for the inductive properties, we need to declare the variable (here, \( j \)) on which induction must be performed. This allows the use of more general induction schemes than induction on the time index\(^4\) (for example induction on processor indices).

PVS theories are generated in turn for these two systems, from the MMALPHA environment. Of course, for such simple properties, proofs are immediate.

6 Conclusion

We have presented a set of methods for proving functional specifications on parallel systems described by means of systems of affine recurrence equations. These methods take advantage of both the computational power of the polyhedral model and the expressivity and verification features of the PVS theorem prover. We are thus able to prove specifications of structured systems in a hierarchical and semi-automatic process.

However, there is still much work to be done. In the current state, the only benefit we gain from the polyhedral model is by using the MMALPHA transformations based on a library of polyhedral computations. We are currently investigating a way to develop specific verification tools adapted to this model (a kind of “polyhedral model-checking”).

\(^4\)Note that the inductive proof performed by using MMALPHA transformations and tautology checking according to the method developed in section 4.1.2 corresponds to a proof by induction on the timing function given by a free schedule, as defined in 5.
Furthermore, we are investigating the possible interactions between refinement (the ALPHA transformation steps) and proof in the two following ways: (i) how can the knowledge of refinement steps lead to specific strategies to save work in the proof process; (ii) how can a certified property enable or disable non-semantics preserving transformations.

References