Verification of Control Properties in the Polyhedral Model

David Cachera, Katell Morin-Allory

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verification of Control Properties in the Polyhedral Model

David Cachera, Katell Morin-Allory

N˚4756

Mars 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4756--FR+ENG

Verification of Control Properties in the Polyhedral Model

David Cachera, Katell Morin-Allory

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n˚4756 — Mars 2003 — 22 pages

Abstract: We propose a combination of heuristic methods to prove properties of control signals for regular systems defined by means of affine recurrence equations (AREs). We benefit from the intrinsic regularity of the polyhedral model to handle parameterized systems in a symbolic way. Despite some restrictions on the form of equations we are able to handle, our techniques apply well for a useful set of properties and led us to discover some errors in actual systems. These techniques have been implemented in the MMALPHA environment.

Key-words: Formal Verification, Automatic Invariant Generation, Verification of Infinite Systems, High-Level Specification Languages, Polyhedral Model

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Verification de propriétés de contrôle dans le modèle polyédrique

Résumé : Nous proposons une combinaison d'heuristiques pour prouver des propriétés sur des signaux de contrôle dans des systèmes réguliers définis à l'aide d'équations récurrentes affines. Nous tirons parti de la régularité du modèle polyédrique pour traiter des systèmesdont la taille est déterminée par des paramètres symboliques. Malgré quelques restrictions sur les formes des équations que nous manipulons, nos techniques s'appliquent à un ensemble significatif de propriétés, et nous ont conduit à découvrir des erreurs dans des systèmes existants. Ces techniques sont implémentées dans l'environnement MMALPHA.

Mots-clé : Vérification Formelle, Génération Automatique d'Invariant,Vérification de Systèmes Infinis, Modele Polyédrique,

1 Introduction

The increasing complexity of embedded systems calls for the elaboration of mechanical, modular and secure codesign tools. The design of a complex and heterogeneous system cannot any more rely on a hand-performed composition of software and hardware components whose interfaces are unformally specified.

The combination of recurrence equations [**?**] over polyhedral domains and affine dependency functions is the basis of the so-called polyhedral model. This model provides a unified framework for expressing hardware and software parts of regular systems. Systems are described in a generic manner through the use of symbolic parameters, and structuration mechanisms allow for hierarchical specifications. The ALPHA language [**?**] and the MMALPHA environment [**?**] provide a syntax and a programming environment to define and manipulate polyhedral equational systems.

Systems are first expressed in a high-level manner, close to the initial algorithmic specification. Then they are refined through a user-guided series of automatic transformations, down to an implementable description, specifying in details how, where and when each computation should take place. From this low-level description, we may derive either loop nest code or an architectural description in VHDL. When the system is partitioned into hardware and software components, interfaces are generated to control communication between both parts [**?**]. For hardware components and interfaces, control signals are generated to validate computations or data transfers.

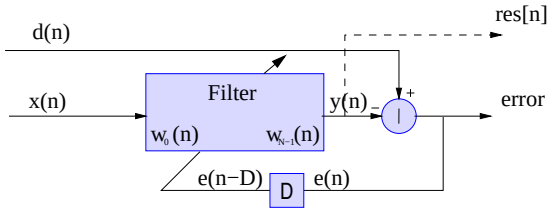
The use of systematic and semi-automatic rewritings together with the clean semantic basis provided by the polyhedral model should ensure the correctness of the final implementation. Nevertheless, the development of real-size systems results in a loss of confidence in the initial high-level specification. 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.

Our aim is to develop proof methods for control properties of systems expressed in the polyhedral model. We thus only deal with boolean signals, but our systems are parameterized. In this paper, we take advantage of the regularity of the polyhedral model, which provides a kind of abstraction to deal with symbolic parameters. Existing tools for manipulating polyhedra will allow us to "hide" inductive proofs behind syntactic substitutions, while the analysis of affine or uniform dependencies will help us in automatically finding invariants. This combination of polyhedra manipulations and automatic invariant searching has been implemented in the MMALPHA environment and proves successful for most properties of parameterized systems described at bit level.

The rest of this paper is organized as follows. In section 2, we show on two very simple examples the kind of properties we want to prove. In section 3 we briefly present the polyhedral model. Section 4 is devoted to a substitution method that allows one to perform inductive proofs without explicitly expressing the induction step. In section 5, we show how to look for invariants in more complex cases, while Section 6 explains how and when we combine these two methods. In section 7, we come back to our examples to show how our proof method works. Section 8 gives an overview of related work, before we give concluding remarks in Section 9.

2 Motivating examples

To motivate our problematics, let us introduce two simple concrete examples. The first example deals with an adaptive filter ([**?**]). The system (*cf.* Fig. 1) consists of N cells, each one containing a weight stored in a register. These cells are organized in 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, 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. The internal generated signal *Wctl1P* is used to control the access to registers $w[i]$. This signal is defined on domain $\{p \leq t \leq p - N + M; 0 \leq p \leq N - 1\}$, where parameters N, M, D are defined on domain $\{N, M, D | 3 \leq N \leq \min(M - D - 1, D - 1)\}$. As long as the system is in its initialization phase, due to the feedback delay, weights are not correctly defined, so *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 registers containing weights. This is expressed by the equation displayed in Fig. 1(b).



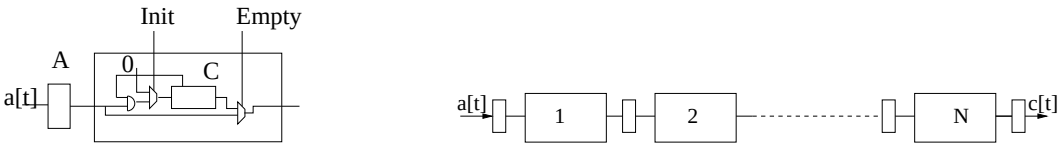
(a) General structure

$$
\begin{aligned}
\mathit{Wctl1P}[t, p] = \\
&\{t, p \mid t \leq D - 1; p = 0\} : \mathit{False} \\
&\{t, p \mid D <= t; p = 0\} : \mathit{True} \\
&\{t, p \mid 1 <= p\} : \mathit{Wctl1P}[t - 1, p - 1]
\end{aligned}
$$

(b) Equation defining an internal control signal, on domain $\{p \leq t \leq p - N + M; 0 \leq p \leq N - 1\}$.

Figure 1: An adaptive filter.

The second example is taken from a system computing a matrix product (*cf.* Fig. 2). In order to simplify the presentation, we only consider here a reduced part of the system, but sharing the same



(a) One cell (b) The array

Figure 2: Structure of an array computing a matrix product.

global structure. The system consists of a linear array of N cells, separated one from another by a register A. Each cell has one data (a) and two control inputs (*Init* and *Empty*), and one data output (c). A cell is composed of one register C in which the local result of each computation is stored. Control signal *Init* is used to initialize register C, and signal *Empty* to empty it. Register A is a delay register. As before, our problem is not to know what the result precisely is, but to ensure that it always has a significant value. More precisely, inputs are entering the array with the following scheme: N matrix coefficient values are input (N being the matrix size), followed by N "empty" values, until the whole matrix has been handled. On the output, the initial N matrix coefficients first get out, followed by N sums of these coefficients. On this particular example, we want to prove that the output is always significant.

Since we are just interested in the presence or not of a significant result, we model integer variables by boolean variables. An instance of a variable is true if the corresponding integer instance is significant, it is false otherwise. Furthermore, we model operations on integer variables by the conjunction of the corresponding boolean operands: the result of an integer operation is significant iff all the operands are significant. This modeling gives us the system displayed in Fig.3.

Here, output c cannot be true before the first coefficient is used in the last cell. The precise property we prove is the following: for all t in $\{t|N < t\}$, $c[t]$ is true.

In both examples, we are not interested in proving a full functional specification of the system, but rather in extracting and automatically checking a simple property concerning control signals that might have been introduced by hand. In the following, we only deal with safety properties, i.e., we will always try to prove that a signal is true on a given domain.

3 The Polyhedral Model

We now present the basic concepts underlying the polyhedral model. We define the notion of System of Affine Recurrence Equations (SARE), then we introduce the notions of schedule, of syntactic transformation of a SARE, and give brief insight to their semantics. 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.

```
system matmult: {N|N>1}
input  (a ,b: {i|0<=i} of boolean)
output (c: {i|1<=i} of boolean);
localvar empty:{t,i|0<=t;1<=i<=N} of boolean;
      init:{t,i|0<=t;1<=i<=N} of boolean;
      C:{t,i|0<=t;1<=i<=N} of boolean;
      A: {t,i|0<=i<=N;0<=t} of boolean;
let
C[t,i]={t,i|t=0}:init[t,i]
       {t,i|t>=1}:init[t,i] or
                  (C[t-1,i] and A[t-1,i-1])
A[t,i]={t,i| t=0; i>=1}:False
       {t,i|t>0;i>=1}:(empty[t,i] and C[t-1,i])
              or (not empty[t,i] and A[t-1,i-1])
       {t,i|t>=0;i=0}:a[t]
init[t,i]={t,i|0<=t;i=1}:b[t]
          {t,i|i>1;t=0}:False
          {t,i|i>1;t>=1}:init[t-1,i-1]
empty[t,i]={t,i|0<=t<=N-1;i=1}:False
           {t,i|N<=t;i=1}:init[t-N-1,i]
           {t,i|i>1;0<=t<2}:False
           {t,i|i>1;t>=2}:empty[t-2,i-1]
c[t] = A[t,N]
tel;
```

Figure 3: Modeling matrix product system

3.1 Polyhedral Domains and Recurrence Equations

Definition 3.1 (Polyhedral Domain) *An n-dimensional polyhedron is a subset $\mathcal{P}$ of $\mathbb{Q}^n$ bounded by a finite number of hyperplanes. It can be implicitly defined by: $\mathcal{P} = \{x \in \mathbb{Q}^n | A.x \leq b\}$ where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$.*

We call polyhedral domain *(or, for short,* domain*) a subset $\mathcal{D}$ of $\mathbb{Z}^n$ defined by*

$$\mathcal{D} = \{x \in \mathbb{Z}^n, z \in \mathcal{P}\} = \mathbb{Z}^n \cap \mathcal{P}$$

where $\mathcal{P}$ is a finite union of n-dimensional polyhedra.

Definition 3.2 (Variable) *A* variable X *is an application from a n-dimensional domain into a base set (booleans, integers or reals); it is said to be an n-dimensional variable. We call* instance *of the variable X any restriction of X to a single point z of its domain, and denote it by $X[z]$. Constants are associated to the trivial domain $\mathbb{Z}^0$.*

Definition 3.3 (Recurrence Equation) *A* Recurrence Equation *defining a function (variable) X at all points, z, in a domain, $\mathcal{D}_X$, is an equation of the form*

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

where

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

  Such an equation may also be written with the following notation.

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

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

$$X[z] = \left\{ \begin{array}{ccc} & \vdots & \\ \mathcal{D}_{X_i} & : & g_i(\dots X[d_i(z)] \dots) \\ & \vdots & \end{array} \right. \tag{1}$$

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 3.4 (Affine Recurrence Equation) *A recurrence equation as defined above, is called an* Affine Recurrence Equation (ARE) *if every dependency 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. If matrix* A *is the identity matrix, dependency* d *is said to be* uniform*. If all the dependencies are uniform, the equation is said to be a* Uniform Recurrence Equation (URE)*.*

Definition 3.5 (System) *A* system *of recurrence equations is a set of* m *equations like (1) , defining the data variables* $X_1 \dots X_m$*. Each variable,* X_i *is of dimension* n_i*, and since the equations may now be mutually recursive, the dependency functions* d *must now have the appropriate type.*

3.2 Dependency Graphs and Schedules

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. Recurrence equations define dependencies between elementary computations, thus constraining the set of possible valid orders.

Definition 3.6 (Dependency Graph) *We say that an instance x of a variable X depends on an instance y of a variable Y if there exists an equation of the form $x = g(\dots, y, \dots)$. We denote by $x \to y$ this fact.*

We call dependency graph *the graph the vertices of which are the variable instances, and where there is an edge between vertices x and y iff $x \to y$.*

We call reduced dependency graph *the graph the vertices of which are the multi-dimensional variables; there is an edge from variable X to variable Y (denoted by $X \to Y$) iff there exists an instance x of X and an instance y of Y such that $x \to y$.*

Definition 3.7 (Schedule) *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 $Y[z']$,*

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

A set of recurrence equations is said to be schedulable *if its dependency graph contains no cycle and if a valid schedule exists. The problem of determining if a SARE is schedulable is undecidable in the general case (see [**?**]), but much work has been devoted to the development of heuristics to find schedules respecting particular formats.*

3.3 System Transformations

The combination of polyhedral domains and affine dependency functions is the basis of the so-called *polyhedral model*. Closure properties in this model (namely by intersection, finite union and inverse of an affine application) are the key of a rich set of formal correctness preserving transformations. The most important manipulation that we can perform on an SARE is called a *reindexing transformation* (also called a *change of basis* or *space-time mapping*) to its variables. This transformation must admit a *left inverse* for all points in the domain of the variable. Let a variable X of a SARE be defined as follows:

$$X[z] = \left\{ \begin{array}{lcl} & \vdots & \\ \mathcal{D}_{X_i} & : & g_i(\dots Y[d_i(z)] \dots) \\ & \vdots & \end{array} \right.$$

Applying reindexing transformation $\mathcal{T}$ to X consists in applying:

- Replace each $\mathcal{D}_{X_i}$ by $\mathcal{T}(\mathcal{D}_{X_i})$.
- On the rhs of the equation for X, replace each dependency d_i by $d_i \circ \mathcal{T}^{-1}$, the composition of d_i and $\mathcal{T}^{-1}$.
- In all occurrences $X[g(z)]$ on the rhs of *any* equation, replace the dependency g by $\mathcal{T} \circ g$.

The SARE we obtain after this transformation is provably equivalent to the original.

From now on, we only consider schedulable systems. Moreover, for a given system, we consider that a schedule has been determined, and that it has been *applied* to the system, i.e., a change of basis has been performed on all variable domains and dependencies to identify for each variable a particular index (say, the first one), which will then be considered as the *temporal index*. Otherwise specified, this particular index will be denoted by t. Such a system will be called a *scheduled system*.

3.4 Semantics

We now give a brief insight on the semantics of SAREs. We use the denotational semantics defined in [**?**] (see this reference for details). As we focus on control properties, we only deal with boolean variables and expressions. The base semantic domain will thus be the following lattice.

$$\mathbb{B} = \{tt, ff, \bot, \top\}$$

where these elements respectively represent true, false, the undefined value (minimum of the lattice) and the erroneous value (maximum of the lattice).

A *multidimensional value* is a mapping from an index space to the base scalar values:

$$VAL = \mathbb{Z}^n \to \mathbb{B}$$

An *environment* ρ maps a variable to a multidimensional value:

$$\rho \in EV = VAR \to VAL$$

The semantics $\mathcal{E}(e)$ of an expression e maps an environment to a multidimensional value:

$$\mathcal{E}(e) \in EXP = EV \to VAL$$

It is defined by induction on the expression's structure.

The semantics of an equation maps an environment to an environment and is defined by

$$\mathcal{Q}(X = e)(\rho) = \rho(sup(\rho(X), \mathcal{E}(e)(\rho))/X)$$

where sup is the least upper bound in $\mathbb{B}$. Finally, the semantics of a system of equations is the least fixpoint of the composition of the semantics of all equations.

An environment that is correct w.r.t. the semantics of a given SARE will be called semantically correct. More formally, we give the following definitions.

Definition 3.8 (Validity) *Let ρ be an environment, e an expression and z an index in the definition domain $\mathcal{D}_e$ of e. We say that ρ valids $e[z]$, denoted by $\rho \models e[z]$, iff*

$$\mathcal{E}(e)(\rho)[z] = tt$$

By extension, we say that ρ valids e on subdomain $\mathcal{D}$ (denoted by $\rho \models_{\mathcal{D}} e$) iff

$$\forall z \in \mathcal{D},\ \rho \models e[z]$$

Definition 3.9 (Well-defined environment) *We say that an environment ρ is* well-defined *for a variable X if for any index z in its domain, $\rho(X)[z] \neq \bot$.*

Definition 3.10 (Semantically correct environment) *We say that an environment is* semantically correct *if it is well-defined and it is a least fixpoint for the system's semantics.*

Definition 3.11 (Initial Points) *We call* initial points of X*, the set of multidimensional indices z in $\mathcal{D}_X$ such that all the variable instance on which $X[z]$ depends are not instances of X:*

$$\mathcal{PI}_X = \{z \in \mathcal{D}_X \mid (\exists z', X[z] \to Y[z']) \implies Y \neq X\}$$

3.5 The ALPHA language and the MMALPHA environment

The ALPHA language [**?**], originally developed for the design of regular (systolic) arrays, provides a syntax to define and manipulate SAREs. An ALPHA program is a mapping from input (polyhedral) variables to output variables defined by a SARE relating input, output and local variables. Each point in the domain of a local variable is uniquely defined by an affine recurrence equation.

The MMALPHA environment [**?**], based on Mathematica, 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, sequential code or VHDL generation.
- Transformations of ALPHA programs, based on polyhedra manipulations and including pipelining of variables, scheduling, change of basis, etc.

4 Proving with substitutions

Let X be a multidimensional variable. Recall that our aim is to prove that, for a given set of values of its initial points, X will be true on its entire domain. As we only consider systems which have been scheduled, we can proceed by induction on time. For any two instances $X[z]$ and $X[z']$ such that $X[z] \to X[z']$, we know that $X[z']$ will be computed before $X[z]$ in the given schedule. We then might assume that $X[z']$ is true in order to prove that $X[z]$ is true. Intuitively, this allows us to substitute $X[z']$ by the constant *True* in the expression defining X. After this substitution step, the resulting expression is simplified according to a set of boolean simplification rules that preserve the semantics of expressions. In the following, we formalize this substitution principle and prove its validity.

4.1 Removing negations

Before performing any substitution, and in order to unify the format of expressions we will work on, we push negations down to input variables: if Z is a variable preceded by a negation in the rhs of an

equation, we temporarily set $Z' = \neg Z$. We replace Z by its definition in equation $Z' = \neg Z$. We then replace in any equation each occurrence of $\neg Z$ by Z' and each occurrence of $\neg Z'$ by Z.

As an example, if Z is defined by

$$Z = \neg Z.d_1 \vee X.d_2$$

We set $Z' = \neg Z$ and $X' = \neg X$, and we obtain

$$\begin{array}{rcl} Z & = & Z'.d_1 \vee X.d_2 \\ Z' & = & \neg Z'.d_1 \wedge \neg X.d_2 \\ Z' & = & Z.d_1 \wedge X'.d_2 \\ X' & = & \neg X \end{array}$$

If X is defined by an equation (i.e., X is not an input variable), we repeat the same manipulation on the expression defining X. We iterate this process until all negations only precede input variables.

4.2 True-Substitutions

Let us now come back to substitutions.

Lemma 4.1 (Substitution) *Let ρ be a well-defined environment and g an arbitrary boolean function. Thus, if ρ valids $X.d$ on a domain $\mathcal{D}$, expressions $g(\ldots, X.d_i, \ldots)$ and $g(\ldots, True, \ldots)$ are semantically equivalent for the environment ρ on domain $\mathcal{D}$.*

Proof. *The proof is immediate, by definition of the semantics of expressions.* ❑

Let e be the expression $g(\ldots, X.d, \ldots)$ in which every instance of the variable X has been substituted by the constant *True*. We define a new variable X' by the following equation:

$$X'[z] = \left\{ \begin{array}{rcl} \mathcal{PI}_X & : & X \\ \mathcal{D}_X \backslash \mathcal{PI}_X & : & e \end{array} \right. \tag{2}$$

Proposition 4.2 *Let ρ be a semantically correct environment, and X' be defined as above. Then*

$$\rho \models_{\mathcal{D}_X} X' \Longleftrightarrow \rho \models_{\mathcal{D}_X} X$$

Proof. *The proof is made by strong induction on the temporal index t_z of z, where z is in $\mathcal{D}_X$. Let t_0 be a fixed value for the temporal index. Domain D_{t_0} is a restriction of domain D_X to those points whose value of temporal index is lower than t_0:*

$$\mathcal{D}_{t_0} = \{z | t_z \leq t_0 \wedge z \in \mathcal{D}_X\}$$

Firstly, we notice that the sequence $(\mathcal{D}_t)_{t \in \mathbb{N}}$ of domains is increasing and converges to $\mathcal{D}_X$. Our induction hypothesis is the following.

$$\forall z \in \mathcal{D}_{t_0},\ \rho \models_{\mathcal{D}_X} X'[z] \Longleftrightarrow \forall z \in \mathcal{D}_{t_0},\ \rho \models_{\mathcal{D}_X} X[z]$$

- *For $t_0 = 0$, domain D_0 is a subset of $\mathcal{PI}_X$. By definition of X', we know that on domain $\mathcal{PI}_X$, $X' = X$. Equivalence is trivial.*
- *We assume that our induction hypothesis is true for any value $t \leq t_0$, we must show that the following equivalence holds.*

$$\forall z \in \mathcal{D}_{t_0+1}, \rho \models X'[z] \Longleftrightarrow \forall z \in \mathcal{D}_{t_0+1}, \rho \models X[z]$$

Let z_0 be a point of $\mathcal{D}_{t_0+1} \backslash \mathcal{D}_{t_0}$, then z_0 is either in $(\mathcal{D}_{t_0+1} \backslash \mathcal{D}_{t_0}) \cap \mathcal{PI}_X$ or in $\mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X)$:

- *If z_0 is in $(\mathcal{D}_{t_0+1} \setminus \mathcal{D}_{t_0}) \cap \mathcal{PI}_X$: by definition of X' on this domain, we have $\rho(X') = \rho(X)$.*
- *Let z_0 be in $\mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X)$. On this domain, we know that $X[z_0]$ is defined by the following equation:*

$$X[z_0] = g(\dots, X[d_i(z_0)], \dots)$$

As we work on a scheduled system, $d_i(z_0)$ is in $\mathcal{D}_{t_0}$. Let us now assume that $\forall z \in \mathcal{D}_{t_0+1}, \rho \models X'[z]$, in particular, we have $\forall z \in \mathcal{D}_{t_0}, \rho \models X'[z]$. By our induction hypothesis, we know that $\forall z \in \mathcal{D}_{t_0}, \rho \models X[z]$. This in particular applies to $d_i(z_0)$. As a consequence, for any z in $\mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X)$, we have $\rho \models X[d_i(z)]$. Due to lemma (4.1) on substitutions, expression $g(\dots, X.d_i, \dots)$ is semantically equivalent to $g(\dots, True, \dots)$. Since ρ is semantically correct, and by definition of X', we can conclude that

$$\begin{array}{l} \forall z \in \mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X), \rho \models X'[z] \Longrightarrow \\ \forall z \in \mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X), \rho \models X[z] \end{array}$$

Conversely, if we assume that $\forall z$ in $\mathcal{D}_{t_0+1}$, $\rho \models X[z]$, we prove in the same way that

$$\begin{array}{l} \forall z \in \mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X), \rho \models X'[z] \\ \Longleftarrow \forall z \in \mathcal{D}_{t_0+1} \backslash (\mathcal{D}_{t_0} \cup \mathcal{PI}_X), \rho \models X[z] \end{array}$$

❑

4.3 Propositional simplification

After true-substitution, we get boolean expressions containing constants. In order to simplify them, we translate expressions into the Mathematica format and rely on the propositional simplification procedures of this tool. The resulting expressions are either single *True* or *False* constants, or boolean expressions involving other variables than X. These expressions are translated back into the ALPHA format.

Of course, this transformation is semantically correct: if e is a boolean expression and e' is obtained from e after propositional simplification, then for any environment ρ and for any z in $\mathcal{D}_e$, $\rho \models e[z] \Longleftrightarrow \rho \models e'[z]$.

4.4 Soundness

From the results of previous sections, we directly can state the following theorem, that expresses the fact that X is true on its entire domain as soon as it is true on its initial points and its defining expression can be reduced to tautologies.

Theorem 4.3 (Soundness of the substitution principle) *Let X be a variable such that true-substitution and propositional simplification have been performed on each expression appearing in the rhs of the equation defining X. If on $\mathcal{D}_X \backslash \mathcal{PI}_X$ X is now defined by tautologies, then for any semantically correct environment ρ such that $\rho \models_{\mathcal{PI}_X} X$, we have $\rho \models_{\mathcal{D}_X} X$.*

4.5 Application to the DLMS Filter

Let us come back to the first example. Recall that variable $Wctl1P$ is defined by the following equation on domain $\{t, p | p \leq t \leq p - N + M; 0 \leq p \leq N - 1\}$.

```
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. The first one is that on domain $\{t, p | p \leq t \leq p + D; 0 \leq p \leq N - 1\}$, *Wctl1P* is false, the second one is that on domain $\{t, p | p + D \leq t < p - N + M; 0 \leq p \leq N - 1\}$, *Wctl1P* is true. For the first property, we work on the negation of Wctl1P. Equations which are defined on the considered domain are selecting by using the polyhedral library. Then the substitution tool automatically yields tautologies. The second property is handled in the same way. All this process is fully automatic.

5 Automatic Invariant Detection

The substitution mechanism succeeds when X depends only on itself (or on initial values), or when the other variables X depends on have been eliminated by propositional simplification rules. This is the case for many simple control signals, for instance when the signal is propagated without many modifications. In that case, X can be seen as an invariant: if X is true "at the beginning", then X remains true "always and everywhere". For more complex control signals, we will have to look for more specific invariants. To restrict the set of formulae we investigate, we try to find an invariant by combining subexpressions appearing in the definitions of variables.

5.1 Determining a set of candidates

Let us focus on a variable W defined by the following equation[1].

$$W = \bigvee_i X_i.d_i$$

We replace in the rhs of this equation each variable by its defining expression, and put the resulting expression into conjunctive normal form. Let us call primary pattern this expression. The idea is to find a variable whose defining expression matches this primary pattern or one of its subterms.

[1] If W is defined by a conjunction, we just have to split it into separate subexpressions and treat them independently.

Let T_W be the set of all disjunctions in the primary pattern, and τ be an element of T_W. Let $Var(\tau)$ be the set of all pairs (V, d) where V is a variable and d a dependency, such that $V.d$ appears in the definition of τ. Let $Used(X)$ be the set of all pairs (V, d) where V is a variable and d a dependency, such that $V.d$ appears in the definition of X.

We select a list $\mathcal{C}(\tau)$ of variables whose defining expressions are "candidates" to match τ: for any variable X, the variables appearing in the defining expression of X must also appear in t; furthermore, there must be an injective application φ from $Used(X)$ to $Var(\tau)$, mapping any pair of $Used(X)$ to another pair in $Var(\tau)$ such that the first component in both pairs is the same.

$$\begin{aligned}\mathcal{C}(t) = \quad & \{X \mid \exists \varphi : Used(X) \hookrightarrow Var(\tau), \\ & \forall (V, d) \in Used(X), \exists d', \varphi(V, d) = (V, d')\}\end{aligned}$$

Let us fix a variable X in $\mathcal{C}(\tau)$. In the following, φ will denote a particular injection valid for X. By extension, we denote by φ, the application mapping d to d' when $\varphi(V, d) = (V, d')$.

Since X is in $\mathcal{C}(\tau)$, X is candidate to match t. We must find a dependency d such that $X.d$ matches a subexpression of τ. That is, we have to find a dependency d'' such that for each (V, d) in $Used(X)$, $d \circ d'' = \varphi(d) \circ d$. Let us denote by $\mathcal{S}(\varphi)$, the set of dependencies such that X matches τ. We will have to solve the following system.

$$\mathcal{S}(\varphi) \quad = \quad \cap_{(V,d) \in Used(X)} \{d'' \mid d \circ d'' = \varphi(d) \circ d\} \tag{3}$$

To find all dependencies that can match τ, we do the union of $\mathcal{S}(\varphi)$ for all possible injections. Notice that all the dependencies in $\mathcal{S}(\varphi)$ have the same dimensions. Since (V, d) is in $Used(X)$, the expression $V.d$ has the same dimension as X. As a consequence, all the matrices representing the second component of a pair in $Used(X)$ have the same number of columns. We can conclude that every element of (3) shares the same number of rows. Similarly, as all the matrices representing $\varphi(d)$ shares the same number of rows, every element of (3) shares the same number of columns.

5.2 Solving a dependencies system

Solving one equation $d \circ d'' = \varphi(d) \circ d$ in (3) is equivalent to finding a matrix X such that $AX = B$, where A and B are known matrices. As we deal with integer matrices, X must be solution to a linear Diophantine equation [**?**]. The Polyhedral Library [**?**] gives an implementation of an algorithm solving such a system, based on the use of Hermite Normal Forms. Intersecting the solutions of a set of such systems is then done by constructing a compound matrix from the matrices of each equation (see [**?**] for more details).

5.2.1 Solving for a vector

We first solve the problem of computing a solution of $Ax = b$ when x and b are integer vectors.

Proposition 5.1 *Let $(A \mid b)$ be a matrix made up of a matrix A lined on right with a vector b. Let $(A' \mid b')$ be the Hermite Normal Form [**?**] of the matrix $(A \mid b)$, and the matrix R which verifies $(A \mid b)R = (A' \mid b')$. A solution to $Ax = b$ is the sum of a particular solution and a vector in the kernel of A: $x = \beta + \Gamma.\xi$*

- *The system $Ax = b$ has a solution if and only if the last row of R is $(0, 0, \ldots, 0, 0, 1)$ and if the vector b' is null. These two conditions give us a particular solution $-\bar{R}_{m+1}$, where $-\bar{R}_{m+1}$ is the vector made with the m first components of R_{m+1}.*
- *In this case, A' is a column echelon matrix left-equivalent to A. Let $k \in [1, m]$ such that the null columns of A are exactly those whose index is greater than k ; then the vectors $\bar{R}_{k+1}, \bar{R}_{k+2}, \ldots, \bar{R}_{m+1}$ are a basis of the kernel of A.*

5.2.2 Solving for a matrix

We have seen how to solve a system $Ax = b$ where b is a vector, but we will have to solve a system of the form $AX = B$ where B is a $n \times p$ matrix, A a $n \times m$ and X a $m \times p$. By juxtaposition of vectors, and as the general part of the solution is always the same (namely, the kernel of A), we immediately have the following proposition.

Proposition 5.2 *If the matrix X is solution of a system $AX = B$, then X can be written $X = \beta + \Gamma.\xi$ where β is a particular solution, columns of Γ are a basis of the kernel of A, and ξ is any matrix of dimension $k \times p$ where k is the dimension of the kernel of A.*

5.2.3 Intersecting solutions

We have found a solution for one equation $d_i.d'' = \varphi(d_i).d_i$. We now must compute a common solution for all the dependencies. We intersect all sets of solutions by solving a system $\Gamma\xi = \beta$ where Γ, ξ, and β are integer matrices.

For the intersection of two sets of solutions S_1 and S_2, Γ, ξ, and β are defined the following way. Let $S_1 = \{\beta_1 + \Gamma_1\xi_1 \mid \xi_1 \in \mathbb{Z}^{k\times p}\}$ and $S_2 = \{\beta_2 + \Gamma_2\xi_2 \mid \xi_2 \in \mathbb{Z}^{k'\times p}\}$. Matrices β_1 and β_2 are in $\mathbb{Z}^{n\times p}$, ξ_1 is in $\mathbb{Z}^{n\times k}$ and ξ_2 in $\mathbb{Z}^{n\times k'}$. The intersection of S_1 and S_2 is $\{\beta_1+\Gamma_1\xi_1 \mid \exists\xi_2, \beta_1+\Gamma_1\xi_1 = \beta_2 + \Gamma_2\xi_2\}$. So we need to find $\xi_1 \in \mathbb{Z}^{k\times p}$ and $\xi_2 \in \mathbb{Z}^{k'\times p}$ such that $\beta_1 + \Gamma_1\xi_1 = \beta_2 + \Gamma_2\xi_2$. In other words, we look for a solution of

$$\beta_1 - \beta_2 = \Gamma_2\xi_2 - \Gamma_1\xi_1 \tag{4}$$

Let $(\Gamma_2 \mid -\Gamma_1)$ be the matrix made up of the matrix Γ_2 lined on right by matrix $-\Gamma_1$, and let $\left(\frac{\xi_1}{\xi_2}\right)$ be the matrix made up of the matrix ξ_2 lined up at the bottom by matrix ξ_1. So, Equation 4 can be rewritten as

$$(\Gamma_1 \mid -\Gamma_2).\left(\frac{\xi_1}{\xi_2}\right) = \beta_2 - \beta_1$$

We generalize to the case $n > 2$ by recurrence on n.

6 Iterating the process

In the two previous sections, we saw two basic operations: true-substitution and invariant computation. We will iteratively combine these two operations. Two cases occur for the considered variable: either it is recursively defined, or it only depends on other variables.

- In the first case, we apply true-substitution. We thus get a variable whose defining expression involves constants (true or false), or boolean expressions depending on other variables. For each branch not defined by a constant, we add a new variable defined on the subdomain of the considered branch by the corresponding expression. This leads us to the second case.
- In the second case, we apply invariant computation (see section 5). If we find an invariant, the variable is recursively defined and we come back to the first case. Otherwise, we will have to repeat the process.

We alternatively repeat both operations, and we stop when, after a true-substitution, either we find a *False* in one branch, or, in every branches, we have an input variable or the constant *True*.

We have to determine in which cases this iteration that alternates true-substitution and invariant computation will stop. For this purpose, we will restrict ourselves to the case of uniform dependencies[2]. In this paper, we will study termination on a base case consisting of variables defined by expressions involving self dependencies, and composed of only one branch. Other cases will be briefly exposed in section 6.2. In the case we study here, we show below that we are able to predict if our iterating process will stop, and if we will find an invariant.

6.1 Case of one variable

We restrict ourselves to the simplest case: we study a variable W defined by an expression depending on only one variable X; this variable X in turn is defined by an expression involving only occurrences of X (self dependencies) and composed of only one branch.

Let W be defined by[3]

$$W = X.d_1 \vee X.d_2$$

At this step, we assume that d_1 is not equal to d_2, as the case $d_1 = d_2$ has been eliminated by true-substitution. Furthermore, we assume that the expression defining X is in conjunctive normal form.

Case of a disjunction Let us first assume that X is defined by a single disjunction

$$X = X.\delta_1 \vee X.\delta_2$$

We substitute X in the definition of W and we get after distribution of "or" operators

$$W = X.\delta_1 \circ d_1 \vee X.\delta_2 \circ d_1 \vee X.\delta_1 \circ d_2 \vee X.\delta_2 \circ d_2$$

Looking for an invariant resumes to finding a solution for one of six systems of equations on dependencies. If such a system has a solution f, we will be able to write $W = W.f \vee X.\delta_i \circ d_j \vee X.\delta_{i'} \circ d_{j'} \vee X.\delta_{i''} \circ d_{j''}$ for some i, i', i'', j, j', j''. After true-substitution, we will get $W = True$. Thus, we just need one solution to stop the iteration. Let us now show that, since we have uniform

[2] In ALPHA, uniformization is used to transform SARE into systems of uniform recurrence equation ([**?**]).

[3] As above, let us remark that W is defined by a disjunction, since for a conjunction we separately study each term of the conjunction.

dependencies, there will always be a solution for at least one of the six systems. The following lemma considers one of the six systems.

Lemma 6.1 *Let d_1, d_2, δ be uniform dependencies. The system*

$$S = \left\{ \begin{array}{lcl} \delta \circ d_1 & = & d_1 \circ f \\ \delta \circ d_2 & = & d_2 \circ f \end{array} \right.$$

has exactly one solution, namely $f = \delta$.

Proof. *Since dependencies are uniform, the product is commutative.* ❑

We can apply this lemma to two out of the six systems (e.g. system S) and we conclude that we are sure to find two invariants. Thus, if X is a disjunction, we are sure to stop the iteration.

Case of a conjunction Let us now assume that X is defined by a conjunction: $X = X.\delta_1 \wedge X.\delta_2$. We substitute X by its definition in W and we get

$$\begin{array}{rcl} W & = & (X.\delta_1 \circ d_1 \vee X.\delta_1 \circ d_2) \\ & & \wedge (X.\delta_2 \circ d_1 \vee X.\delta_2 \circ d_2) \\ & & \wedge (X.\delta_1 \circ d_1 \vee X.\delta_2 \circ d_2) \\ & & \wedge (X.\delta_2 \circ d_1 \vee X.\delta_1 \circ d_2) \end{array}$$

Since W is a conjunction, we now have to find an invariant in each factor if we want to stop the iteration. Thanks to Lemma (6.1), we find an invariant in the first two factors (resp. $W.\delta_1$ and $W.\delta_2$). For the last two factors, we show that we cannot find any invariant using this method. Let W_1 and W_2be respectively defined by $W_1 = X.\delta_1 \circ d_1 \wedge X.\delta_2 \circ d_2$ and $W_2 = X.\delta_2 \circ d_1 \vee X.\delta_1 \circ d_2$. For each variable W_1 and W_2, we have two systems to solve. If we have no solution, we substitute X by its definition and we get (for variable W_1)

$$\begin{array}{rcl} W_1 & = & (X.\delta_1 \circ \delta_1 \circ d_1 \vee X.\delta_1 \circ \delta_2 \circ d_2) \\ & & \wedge (X.\delta_2 \circ \delta_1 \circ d_1 \vee X.\delta_2 \circ \delta_2 \circ d_2) \\ & & \wedge (X.\delta_1 \circ \delta_1 \circ d_1 \vee X.\delta_2 \circ \delta_2 \circ d_2) \\ & & \wedge (X.\delta_2 \circ \delta_1 \circ d_1 \vee X.\delta_1 \circ \delta_2 \circ d_2) \end{array}$$

Again, for the first two factors of W_1, we find an invariant thanks to Lemma (6.1), and the same problem arises for the last two ones. By iterating this process, we get the four systems described in Lemma (6.2) below.

Lemma 6.2 *Let $d_1, d_2, \delta_1, \delta_2$ be uniform dependencies. Let n be a positive integer. Let $S_1^n, S_2^n, S_3^n, S_4^n$ be the four systems defined by the following equations.*

$$S_1^n = \left\{ \begin{array}{lcl} \delta_1^n \circ d_1 & = & d_1 \circ f \\ \delta_2^n \circ d_2 & = & d_2 \circ f \end{array} \right. \quad S_3^n = \left\{ \begin{array}{lcl} \delta_1^n \circ d_1 & = & d_2 \circ f \\ \delta_2^n \circ d_2 & = & d_1 \circ f \end{array} \right.$$

$$S_2^n = \left\{ \begin{array}{lcl} \delta_2^n \circ d_1 & = & d_1 \circ f \\ \delta_1^n \circ d_2 & = & d_2 \circ f \end{array} \right. \quad S_4^n = \left\{ \begin{array}{lcl} \delta_2^n \circ d_1 & = & d_2 \circ f \\ \delta_1^n \circ d_2 & = & d_1 \circ f \end{array} \right.$$

Systems S_1^n, S_2^n have no solution. If system S_3^n has a solution, system S_4^n doesn't and conversely.

Proof. *Since δ_1 and δ_2 are not equal, systems S_1^n and S_2^n have no solution. We now prove the last assertion. Let us suppose that S_3^n has a solution, then we have*

$$\delta_1^n \circ d_1^2 = \delta_2^n \circ d_2^2 \tag{5}$$

Similarly, if S_4^n has a solution, we have

$$\delta_1^n \circ d_2^2 = \delta_2^n \circ d_3^2 \tag{6}$$

By combining equations (5) and (6), we get

$$d_1^4 = d_2^4$$

Since d_1 and d_2 are uniform dependencies,

$$d_1^4 = d_2^4 \Longleftrightarrow d_1 = d_4$$

As a consequence, systems S_3^n and S_4^n cannot simultaneously have a solution. ❑

In the best case, we will find an invariant for one of the two last factors, but for the other the iteration will never stop.

6.2 Other cases

In the case of a variable depending only on itself through uniform dependencies, we have seen that we are able to predict termination of the invariant searching method. Another interesting case is when the variable involved in the invariant is defined by several equations on different domains. In that case, we are able to show that we will always get an invariant, except in one particular case that does not occur in practice and can nevertheless be avoided by a simple system transformation. This case is not detailed here and will be the topic of a further paper.

If the variable involved in the invariant depends on several distinct variables, the problem is similar to the one treated here, and we can predict the existence of an invariant. Schematically, in the case of a conjunction of at least two instances of the same variable, we will not find any invariant, and in the other cases our method will fail in the case of a strongly connected reduced dependency graph. We are currently developing other heuristics to handle these cases.

7 Back to the examples

7.1 Implementation

The methods described below are implemented in MMALPHA, using the Polyhedral Library [**?**] for computations on polyhedra. Substitution and invariant searching are automatically performed, but the user still has to provide the exact domains on which these operations have to be made. The two main functions we use are `iterateChecker` and `proveListOfProperties`. The first one is used to prove just one property, the second to prove a list of mutually dependent properties(like

the proof of periodicity in the example below). Inputs of the `iterateChecker` function are a system, and a *proof node* composed of the name of the property, its defining expression, the domain on which the proof has to be performed, and an integer representing the proof status. Output of this function is the transformed system and a proof tree: the root is the property and the internal nodes are the proof obligations that have been generated to prove the property. The status of the proof is represented by the status value in the root. In case of success, the proof tree leaves contain preconditions that have to be verified by input variables. When the proof fails, these leaves give new proof obligations. The second function `proveListOfProperties` is very similar to the first one, its inputs are the system and the list of properties in proof node format. Properties are proved one after another using at each step the previously transformed system. After having tried one proof, its results are used to simplify the proof trees of previously handled properties.

7.2 Control in a matrix product linear array

Let us come back to the system computing a matrix product displayed in Fig. 2. Recall that the problem is to show that we always have a significant result on the array output, i.e., for all t in $\{t|N < t\}$, $c[t]$ is true. We are given an external specification of the system by means of the following statements.

- $b[t] \Longleftrightarrow \neg a[t-1] \wedge a[t]$,
- b and a are periodic signals,
- their common period is equal to $2N$,
- a is an alternation of N true values and N false values, the first true value being at $t = 1$.

In the following, we first prove that all the signals are periodic, then study c on the first period.

7.2.1 Periodicity

Let $e[t]$ be a periodic signal, of period P: we have $e[t] = e[t-P]$. This can be rewritten in the following equations:

$$\begin{cases} e[t] \vee \neg e[t] = True \\ e[t-P] \vee \neg e[t-P] = True \\ e[t] \vee \neg e[t-P] = True \\ e[t-P] \vee \neg e[t] = True \end{cases}$$

Since the first two equations are tautologies, we can omit them.

For each signal sig in our system, we automatically generate two variables defined by $sigper1[i,t] = sig[t-P,i]\, or \neg sig[t,i]$ and $sigper2[i,t] = sig[t,i]\, or \neg sig[t-P,i]$, and we prove that these two variables are true on their whole domain. As a first step, the system is automatically rewritten without negations.

We just sketch one of the eight proofs. We want to prove that variable $Aper1 = A[t-2N,i]\, or\, notA[t,i]$ is true on Domain $\{t,i|0 <= i <= N; 2N <= t\}$. Since dependencies are

uniform, and since there is just one instance of A in the definition of A, we are sure to find an invariant by using the invariant searching method. After invariant searching, we are left with $Aper1$ depending of $Cper1$, $Aper1$, $emptyper1$, A, $notA$, C, $notC$, $empty$, $notempty$. We repeat the process for every freshly introduced variable and we use the substitution method on every variable. In practice, since the proofs are mutually dependent, we use the `proveListOrProperty` function, after a first execution, we obtained some new proof obligations. The verification of these new proof obligations allow us to simplify initial proof trees, thus concluding the proof.

7.2.2 Considering one period

We now know that all signals are periodic. We just have to prove that $c[t]$ is true for $t \in \{t, N | N < t \leq 3N\}$. We apply the substitution method and the problem is automatically turned into proving that $A[t, i]$ is true on domain $\{t, i, N | N < t \leq 3N; i = N\}$. At this point, the proof cannot be achieved, because the considered domain has to be widened. We use a heuristic procedure that takes the different dependencies into account to propose a way to widen the domain. The proof is restarted with this enlarged domain, yielding new proof obligations on other variables, which are handled the same way. There is one more case were the domain has to be widened (namely for variable C). At the end of the proof process, we get a precondition on input variables that must be initially ensured if we want c to be true on the considered domain. In this particular example, the precondition we get is the following.

$$b[1] \wedge \forall t \in \{t | 1 \leq t \leq N\}\ a[t]$$

This precondition is a consequence of the external specification [4].

8 Related work

This work uses an approach based on substitutions earlier described in [**?**], where a similar method was used to prove equivalence of two SAREs. In this paper, we generalize this approach to safety properties on a given polyhedral domain, and give a new heuristic method to look for invariants when the simple approach fails.

Though many properties concerning parameterized SAREs are undecidable, recent work gives heuristics for automatically checking equivalence of two SAREs in some particular cases [**?**].

When dealing with parameterized systems, we might either use tools based on higher-order logics like theorem provers, or try to adapt methods initially devoted to finite-state systems. The former approach has been widely investigated in the general framework of formal hardware verification, combining various Hardware Description Languages with various theorem proving tools. In the polyhedral model framework, this approach has been investigated using the PVS proof assistant [**?**]. SAREs are translated into the PVS specification language, and interactive proofs are partially automated through the use of specific strategies [**?**]. A similar work has been done in [**?**], where equivalence of VHDL descriptions (at bit level and at arithmetical level) is proved thanks to the Nqthm

[4] In fact, the initial specification assumed $a[1] = b[0] = True$ and $b[1] = False$, which lead to an error. Moreover, our proof method led to discover an error in the system itself.

theorem prover. Generalization heuristics are used to guess invariants for proving equivalence between recursive functions and iterative ones. This approach only applies to one- or two-dimensional architectures, and requires regularity to be expressed through precise and fixed constructs.

Even if Apt & Kozen have shown that model-checking techniques cannot be applied to regular networks of processes of unknown size [**?**], it is still possible to adapt this kind of techniques to parameterized systems in some particular cases. Many authors restrict themselves to the case of linear networks of processes. In [**?**] for instance, Halbwachs & al. prove mutual exclusion in a linear hardware arbiter. They rely on complex user-given invariants to prove the induction step from one parameter value to the next one. Such an approach is not easily generalizable. In [**?**], a more general approach to linear networks is developed, using observers and network invariants: invariants are expressed as fixpoints, and over-approximations are found by widening techniques. In this kind of tool, the user has to adjust extrapolation parameters. Other approaches inspired from abstract interpretation try to automatically construct a finite approximation of an infinite-state system. In [**?**], boolean abstraction is performed in the framework of guarded transition systems, using predicates over concrete variables as boolean abstract variables. Invariants are generated by static analysis techniques. The whole process consists in a series of refinements leading in case of success to a finite-state abstract description that can be model-checked. In contrast to these approaches, we do not try to define complex invariants to perform inductive proofs on the system size. We rely on the "natural" abstraction of the polyhedral model to handle parameters in a symbolic fashion, and our invariants are "temporal" invariants that focus only on the partial property we want to prove.

9 Conclusion

We have presented heuristic methods for proving control properties of regular parameterized systems described by means of affine recurrence equations. These methods take advantage of the computational power of the polyhedral model and of the way it naturally provides abstraction for generic systems. We combine syntactic substitutions that provide a simple induction principle, and invariant generation in a semi-automatic process.

When restricting ourselves to uniform dependencies and scheduled systems, we are able to identify cases where invariant searching will be useful. In practice, as we deal with low-level descriptions with few complex signal manipulations, these methods will prove successful in most cases. As we have seen in the matrix product example, it is often necessary to extend the polyhedral domain on which we try to prove a given property. In the current implementation, domain widening is performed thanks to a very simple heuristics. We definitely should try to develop more specific widening operators.

Of course, there is still much work to be done. We are currently investigating other ways to find invariants in presence of conjunctions, that are based on the identification of specific patterns in polyhedral lattices. We think that this could also provide a way to extend the class of properties we are able to prove, by considering not only safety properties, but also liveness properties.

Acknowledgments

The authors want to thank Patrice Quinton for fruitful discussions on this topic.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399