A Logical Framework to Prove Properties of Alpha Programs (revised version)

Luc Bougé, David Cachera

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A logical framework to prove properties of ALPHA *programs (revised version)*

Luc Bougé, David Cachera

N° 3177

Juin 1997

THÈME 1

ISSN 0249-6399

INRIA
RHÔNE-ALPES

A logical framework to prove properties of ALPHA programs (revised version)

Luc Bougé*, David Cachera*

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n°3177 — Juin 1997 — 14 pages

Abstract: We present an assertional approach to prove properties of ALPHA programs. ALPHA is a functional language based on affine recurrence equations. We first present two kinds of operational semantics for ALPHA together with some equivalence and confluence properties of these semantics. We then present an attempt to provide ALPHA with an external logical framework. We therefore define a proof method based on invariants. We focus on a particular class of invariants, namely canonical invariants, that are a logical expression of the program's semantics. We finally show that this framework is well-suited to prove partial properties, equivalence properties between ALPHA programs and properties that we cannot express within the ALPHA language.

Key-words: Concurrent programming, recurrence equations, specifying and verifying and reasoning about programs, semantics of programming languages, data-parallel languages, proof methodology, invariants.

(Résumé : tsvp)

This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS.

* Contact: {Luc.Bouge,David.Cachera}@ens-lyon.fr. LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon cedex 07, France.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 76 61 52 00 – Télécopie : (33) 76 61 52 52

Un cadre logique pour la preuve de programmes ALPHA (version révisée)

Résumé : Nous présentons une méthode de preuve par assertions pour les programmes ALPHA. ALPHA est un langage fonctionnel d'équations récurrentes affines. Nous présentons tout d'abord deux types de sémantiques opérationnelles pour ALPHA, ainsi que des propriétés d'équivalence et de confluence de ces sémantiques. Nous munissons ensuite ALPHA d'un cadre logique externe au langage. Nous définissons pour cela une méthode de preuve fondée sur l'utilisation d'invariants. Nous insistons sur une classe particulière d'invariants, les invariants canoniques. Nous montrons finalement que ce cadre permet de prouver des propriétés partielles, des équivalences de programmes et surtout des propriétés non exprimables en ALPHA.

Mots-clé : Programmation parallèle, équations récurrentes, spécification et validation de programmes, sémantique des langages de programmation, langages data-parallèles, méthode de preuve, invariants.

Introduction

The ALPHA programming language has been designed by Quinton and his group [7] as a basis to design, combine, optimize and eventually compile systolic algorithms. An ALPHA program is a set of affine recurrence equations (ARE) on multidimensional polyhedric integer domains. ALPHA is a strict, strongly typed functional language, restricted to a very simple form of mutual recursion, and it has proved to be an valuable tool in this field.

Usual manipulations on systolic algorithms can be seen as textual syntactic rewritings on the text of ALPHA programs. Parallelization amounts to reindexing the equations through a change of basis. Optimization amounts to add, delete and combine equations and variables according to a number of equivalence-preserving rules. Finally, compiling an ALPHA program amounts to eventually rewrite it into such a simple form that each equation corresponds to some elementary logical operator, each indexing to some hardware connection, and one of the component of the underlying basis to the hardware clock.

Much work has therefore been devoted to studying such transformations, and designing tools to assist the user in applying them. Yet, the problem of abstractly *proving* properties of ALPHA programs has been somewhat overlooked, and one can find many "proofs by handwaving" in the literature. The situation is reminiscent of the one prevailing in the 70's and 80's in the field of program verification. Two alternative directions were competing to prove sequential programs correct.

Internal: Refine the program up to equivalence, so that the intended property is eventually made explicit.

External: Equip the program with some kind of logical assertions, so that the proof of correctness boils down to checking their abstract mutual consistency.

For Pascal-like imperative programs, the balance is clearly in favor of the external approach (Floyd's assertions, Hoare's logic, Gries-Owicki method, etc.). No other alternative may develop, as the transformational calculus of such imperative programs is so poor. In contrast, more abstract frameworks such as Unity or CCS present a balance picture: the two approaches co-exist and fruitfully interact.

The ALPHA language benefits from a rich and moreover semi-automatizable transformational calculus, so that the internal approach has been almost exclusively developed. Yet, the external approach is informally used in several papers such as [9] where an ALPHA program is proved to be equivalent to its closed form, and [4] where a logical induction argument is used to prove a refinement step. The goal of this paper is precisely to set up an external framework in which these "handwaving" manipulations can be expressed and justified. Moreover, we show that this framework can give account of a number of methods to prove the equivalence of ALPHA programs, and open new direction in the search for useful internal textual transformations.

The paper is organized as follows. We first discuss more precisely the notion of a proof in such a context, starting from examples found in the literature. Then, we briefly recall the essentials of the ALPHA language for self-containedness, and we discuss its operational semantics as a basis to interpret logical formulae. Then, we define our logical framework and mention its completeness: all correct assertions can be proved. Finally, we show that this framework can give account of some pragmatic methods used so far. The discussion sums up our contribution and lists perspectives.

1 Proving ALPHA programs: What's the problem?

Let us review the problems raised by the proof of Alpha programs, and more generally the proof of programs expressed in a single assignment language.

1.1 Proofs of equivalence

Consider the following ALPHA program computing a convolution.

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | i>=1} of integer)
returns (y : {i | i>=4} of integer);
var
 Y : {i,j | 0<=j<=4 ; i>=4} of integer;
let
  Y[i,j] = case
             {| j=0 } : 0[];
```

```
            {| 1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
          esac;
  y[i] = Y[i,4];
tel;
```

We may wish to prove that this program is equivalent to another one where the $Y[i,j]$ value is computed in a different way. Say for instance:

```
  Y[i,j] = case
            {| j=0 } : 0[];
            {| j=1 } :  a[1] * x[i];
            {| 2<=j<=4} : a[j-1] + Y[i,j-1]
                          + a[j] * x[i-j+1] - a[j-1] ;
          esac;
```

It is likely that wise enough rewriting rules will do the work, but observe that it involves already subtle semantics knowledge, such that $a+b+c-a=a$ which is not always valid in real cases (think of intermediate overflows!).

A more subtle example is to introduce an additional dummy variable, say aux:

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | i>=1} of integer)
returns (y : {i | i>=4} of integer);
var
 Y : {i,j | 0<=j<=4 ; i>=4} of integer;
 aux : {i | i>=4} of integer;
let
  Y[i,j] = case
            {| j=0 } : 0[];
            {| 1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
          esac;
  aux[i] = Y[i,4];
  y[i] = aux[i];
tel;
```

Again, it is likely that any system will be able to reduce this to

```
  y[i] = Y[i,4];
  aux[i] = y[i];
```

but it is *really* the same program as `y[i] = Y[i,4];`? For instance, is `A[] = B[];` *really* the same as `A[] = B[]; C[] = B[]/b;` (think of the case where $b=0$!)?

1.2 Proofs of partial properties

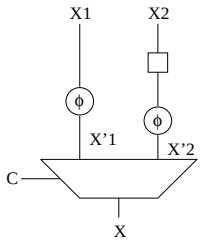
Consider again the convolution program above. Using subtle rewriting, I may eventually convince myself that this program indeed computes a convolution. This will in any cases use heaps of papers, pencils and time. Moreover, in many cases, the property I want to prove is much simpler than such a detailed specification. For instance, I may be only interested in proving that no variable overflows during the computation (think of Ariane 5) and not in their precise values. For instance, if the coefficients are bounded by M and the input values are bounded by K (precondition P), then the output values are bounded by $4KM$ (postcondition Q):

$$\begin{aligned} P &\equiv \forall j:\ (1 \le j \le 4) \Rightarrow |a[j]| \le M \\ &\qquad\qquad \wedge\ |x[i_0+j-1]| \le K \\ Q &\equiv |y[i_0]| \le 4KM \end{aligned}$$

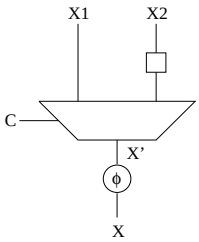
This property may be so explicit in the final transformed program that I may consider it as "obvious". But the amount of work needed to carry out the transformation is very likely to be without any common magnitude with this simple property: simple properties should have simple and rigorous proofs!

A striking example of such a case is given in a paper by Rajopadhye [9] where a timing property is proved on a complex ALPHA program. Using complex (and informal!) transformations, the ALPHA program is "abstracted up" to its uninterpreted computation scheme. Then, this scheme is proved to be equivalent to a function defined by a closed form by replacing the function text in the scheme and rewriting everything to tautologies. Why should it be OK to forget the computational meaning of a program to only consider its operational behavior? Is is always possible? If I succeed proving proving a property on the uninterpreted form, does it always apply to the interpreted one? Can I always prove that an ALPHA program computes a function expressed in closed form by plugging the function in the program and rewriting everything down to tautologies?

1.3 An interesting example




```
-- Program 1 (before retiming)
system hardware_share1: {N|N>=0}
   (X1,X2: {t| t>=0;t<=N+1} of integer;
   C:{t|t>=0} of boolean)
returns (X: {t|t>=0;t<=N+1} of integer);
var
   X1prim: {t| t>=0;t<=N+1} of integer;
   X2prim: {t| t>=1;t<=N+1} of integer;
let
   X[t] = if C[t] then
          case {|0<=t<=N}: X1prim[t];
               {|t=N+1}: 0[];
          esac else
          case {|1<=t<=N+1}: X2prim[t];
               {|t=0}: 0[];
          esac;
   X1prim[t] = (X1[t])*2[];
   X2prim[t] = (X2[t-1])*2[];
tel;
```




```
-- Program 2 (after retiming)
system hardware_share2: {N|N>=0}
   (X1,X2: {t| t>=0;t<=N+1} of integer;
   C:{t|t>=0} of boolean)
returns (X: {t|t>=0;t<=N+1} of integer);
var
   Xprim: {t| t>=0;t<=N+1} of integer;
let
   X[t] = Xprim[t]*2[];
   Xprim[t] = if C[t] then
          case {|0<=t<=N}: X1[t];
               {|t=N+1}: 0[];
          esac else
          case {|1<=t<=N+1}: X2[t-1];
               {|t=0}: 0[];
          esac;
tel;
```

Figure 1: A sample circuit (hardware share) and its retimed version

As a "real-life" example, consider the programs displayed on Figure 1: two streams of input data X_1 and X_2 are to be processed by an operator Φ (multiplication by 2 in these examples) and then interleaved[1]. On the left-hand circuit, there are two copies of the operator. On the right-hand circuit, the operator is shared thanks to retiming.

Proving that the two programs are equivalent can be done by rewriting. Assume now that we specify the C multiplexer as follows:

```
C[t] = case {|t=0}: true;
            {|t>0}: not C[t-1];
```

It is intuitively clear that for even values t, $X[t] = X_1[t]$, and for odd values $X[t] = X_2[t-1]$. Yet, even this very simple property cannot be proved by rewritings:

- It cannot be even expressed: `{|u>=0}: X[2u]=X1[2u]` is not a legal ALPHA program, as the set of even integers is not a polyhedral domain.
- It cannot be proved as it involves some form a induction to prove that $C[t]$ is true on even values of t.

An external logical framework is definitely needed here.

[1] One might notice that some input or local data-fields are defined on "too large" domains: for instance, the values of C for $t > N + 1$ are not necessary to the computation of the ouput. This point will be treated in the next section.

2 An operational semantics for ALPHA

2.1 Basics

An extensive presentation of the ALPHA language can be found in [10]. We just recall here what is useful for our purpose. For the sake of simplicity, we adopt the following definitions and notations.

An ALPHA program is given by:

- a finite set of *data-fields* X on a (possibly infinite) multidimensional polyhedral domain $\mathcal{D}_X$. A component $X[\vec{i}]$ of a data-field is called a *variable*[2];
- for each domain $\mathcal{D}_X$, a disjoint partition into a finite number of polyhedral subdomains
$$\mathcal{D}_X = \mathcal{D}_X^1 \cup \mathcal{D}_X^2 \cup \ldots$$
- for each subdomain $\mathcal{D}_X^p$, a defining equation
$$\forall \vec{i} \in \mathcal{D}_X^p \quad X[\vec{i}] = \varphi(Y[f(\vec{i})], Z[g(\vec{i})], \ldots).$$
- a set of input *variables* In;
- a set of output *variables* Out.

Individual component variables $X[\vec{i}]$ will be denoted x for the sake of conciseness. An ALPHA program is thus a (possibly infinite) set of defining equations $x = \varphi(y, z)$, at most one for each variable x.

Dependence. We say that x directly depends on y if there exists some equation $x = \varphi(y, z)$. We denote by $\preceq$ the reflexive and transitive closure of this dependence relation. A program is well-formed if is has no defining equation for input variables, and if there is no cycle in the dependence relation between variables. *We only consider well-formed programs here.*

States. The set of values contains a special value $\perp$ which denotes undefinedness. A state σ is a mapping from the set of variables to the set of values. The support of the state σ is the set of variables that are valued in σ: $supp(\sigma) = \{x \mid \sigma(x) \neq \perp\}$. We distinguish a particular subset of states, that is the set of *initial* states denoted by *Init*: $\sigma \in Init$ if and only if $supp(\sigma) = In$. We also need to define a notion of final state. In order to avoid problems due to the fact that *Out* may be infinite, a final state is defined with respect to a finite subset V of output variables: σ is final w.r.t. V if and only if all variables of V are valued in σ.

2.2 Two kinds of operational semantics

The definition of ALPHA does not specify the order of evaluation for expressions. A program may have several evaluation paths, whose union forms an execution graph [8]. Each transition in the graph corresponds to the evaluation of *one* single point of a data field, i.e. to one variable.

Two approaches have been proposed in [8] as a basis for an operational semantics of ALPHA programs. They differ in the way the "execution graph" is generated. The first one is the dataflow approach: an expression is evaluated as soon as its free variables have been given a value. The second one is a demand-driven approach: an expression is evaluated when its free variables are known, but only if this evaluation is necessary to the computation of the output variables. These two approaches yield two different transition rules. In both approaches, a transition starts from a given state σ, and yields a state σ' valuating some variable x. The only difference between the two approaches lies in the premise of the transition rules.

We start with the dataflow approach. The only conditions we have to check is that we are able to compute the value of x, i.e. that the variables x is depending on are already valued, and that x has not yet been valued. When these conditions are verified, we just compute the result of the defining expression $\varphi(y, z)$ of variable x, and update σ with the value obtained by this computation.

Rule 1 (dataflow)
$$\frac{(\sigma(y) \neq \perp \wedge \sigma(z) \neq \perp) \wedge \sigma(x) = \perp}{\sigma \xrightarrow{x} \sigma[x \leftarrow \sigma(\varphi(y, z))]}$$

A sequence $(\sigma_i)_{i \in \{0,\ldots,n\}}$ *of states such that* $\forall i \in \{1, \ldots, n\}, \sigma_{i-1} \xrightarrow{x_i} \sigma_i$ *is called a* transition sequence.

[2] In the usual presentation of the ALPHA language, the term "variable" refers to what we call here "data-field".

In the demand-driven approach, we do not consider any longer that an ALPHA program must "consume" its input variables, but that it must "produce" its output variables. In that sense, we restrict computations to those which are necessary for the evaluation of output variables.

Considering a subset V of Out, the set of *necessary* variables w.r.t. V is the set of variables appearing on a path between In and V. This is formalized in the following definition.

Definition 1 (Necessary) *Let V be a subset of Out.*

$$Necessary(V) = \{x \mid \exists z \in In, \exists t \in V,\ z \preceq x \preceq t\}$$

As a convention. $Necessary(Out)$ will simply be denoted by $Necessary$.

In the transition rule, we now have to check that the computation of the variable x is necessary.

Rule 2 (demand-driven)

$$\frac{(\sigma(y) \neq \bot \wedge \sigma(z) \neq \bot) \wedge \sigma(x) = \bot \wedge x \in Necessary(V)}{\sigma \underset{V}{\overset{x}{\Longrightarrow}} \sigma[x \leftarrow \sigma(\varphi(y,z))]}$$

In the case where $V = Out$, we simply write $\sigma \overset{x}{\Longrightarrow} \sigma'$.

2.3 Equivalence between the two approaches

As the second rule is a restriction of the first one, the two operational semantics cannot be strictly equivalent. We thus define a notion of equivalence *modulo* some set of (output) variables. We say that two states σ and σ' are *equivalent modulo V*, which is denoted by $\sigma \underset{V}{\sim} \sigma'$ iff each variable x of V has the same value (possibly $\bot$) in σ and in σ'.It can be shown that, according to the above semantics, all the possible final states of a program are equivalent.

In [2] we show that, from every sequence of transitions obtained by means of the first transition rule, we are able to obtain a *rearranged* sequence where all variables of $Necessary$ are valued first. This is the key to establish the following theorem of equivalence between the two operational semantics, whose proof is given in [2].

Theorem 1 (Equivalence) *Let V be a finite subset of Out, and σ_0 a state in $Init$. If there exists a sequence*

$$\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$$

with $V \subseteq supp(\sigma_n)$, then there exists a sequence

$$\sigma_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{m-1}}{\Longrightarrow}} \sigma'_m$$

with $\sigma_n \underset{V}{\sim} \sigma'_m$ and $\langle x'_0, \ldots, x'_m \rangle$ is a subsequence of $\langle x_0, \ldots, x_n \rangle$. The converse trivially holds.

3 An external logic

We now want to express and prove properties on ALPHA programs. We therefore have to define an assertion language that will be expressive enough for our purpose. We then have to make precise the notions of specification, of validity and of provability for specifications.

3.1 Assertion language

The assertion language we use is defined the following way.

- An index expression is an integer constant, an index variable, or the combination of index expressions with usual arithmetical operators (including modulo).
- An arithmetic expression is a constant value (possibly $\bot$), an integer ALPHA data-field indexed by a tuple of index expressions, or the combination of arithmetic expressions with usual operators. The size of the tuple indexing a data-field corresponds to the dimension of this data-field.

- A boolean expression is a boolean constant, a boolean data-field indexed by a tuple of index expressions, the combination of arithmetic or index expressions with usual relation operators ($=, <$, etc.), or a conditional expression $(b\ ?\ e_1 : e_2)$.
- A formula of the assertion language is the combination of boolean expressions with usual boolean connectives ($\wedge, \vee, \neg, \Rightarrow$), or a quantified formula. The usual existential and universal quantifiers apply on indices in the domain of a data-field. Formulas must be closed, and we assume implicit external quantification.

For instance,

$$\forall t \geq 0, \forall p \geq 0 : X[t,p] = 0$$

means that X is equal to zero on the domain $\mathbb{N} \times \mathbb{N}$;

$$\forall t \geq 0 : X[2*t] = \mathit{true} \wedge X[2*t+1] = \mathit{false}$$

means that $X[t]$ is true for even values of t, and false otherwise.

For the sake of completeness, the interpretation of arithmetic or relational operators has to take into account the presence of the $\perp$ value that is explicitly manipulated in the semantics. Arithmetic expressions are extended in a strict way: the evaluation of an expression in a state where one or more of the variables necessary to this evaluation is undefined yields the undefined value. For instance, a conditional expression $(b\ ?\ e_1 : e_2)$ is undefined if one or more of its arguments is undefined. For each relational operator, we assume that a suitable extension has been defined. The only requirement is that the standard interpretation of equality is preserved: $\perp = \perp$ is true.

Our specification language explicitly manipulates the $\perp$ value. This enables us to write formulas like $X[i] = \perp$. We therefore add a new "equality modulo definedness" abbreviation, denoted by $\stackrel{\Delta}{=}$. Formula $a \stackrel{\Delta}{=} b$ is simply a shorthand for

$$a \neq \perp \Rightarrow (a = b).$$

It is true if either its left-hand side is undefined, or both sides are defined and have the same value. We also introduce a special abbreviation Δ for definedness.

$$\Delta(x) \equiv \neg(x = \perp)$$

The interpretation of the other predicates, boolean connectives or quantifiers is standard and intuitively straightforward. If σ is a state and P a formula of the assertion language, we denote by $\sigma \models P$ the fact that the interpretation of P is true in σ.

As an example, $\sigma \models (X[i] \stackrel{\Delta}{=} Y[j])$ for $\sigma = \{X[i] = \perp, Y[j] = \perp\}$ or $\sigma = \{X[i] = \perp, Y[j] = 2\}$ or $\sigma = \{X[i] = 2, Y[j] = 2\}$, but $\sigma \not\models (X[i] \stackrel{\Delta}{=} Y[j])$ for $\sigma = \{X[i] = 2, Y[j] = \perp\}$.

3.2 Validity of a specification

As we aim at giving Alpha an assertional semantics inspired from the Hoare logic for imperative languages, we have to define formulas like $\{P\}\ S\ \{Q\}$, where S is a program and P, Q are two assertions. Such a formula means that, if we start from a state satisfying precondition P, the execution of S yields a state satisfying postcondition Q if any final state is reached.

The notion of initial state is not so difficult to define: we take states $\sigma \in \mathit{Init}$. But, because of the "dataflow" nature of the language, it may happen that all variables necessary to the result have been valued and some additional equation remains firable. As a consequence, the notion of terminating program and of final state are not that clear. Of course, we only consider postconditions Q which do not depend on these extra firing. This is formalised in the following definition.

Definition 2 (Stability) *We say a predicate Q is stable w.r.t. V iff its validity is the same for all states equivalent modulo V: if $\sigma \models Q$ and $\sigma \underset{V}{\sim} \sigma'$, then $\sigma' \models Q$. A specification $\{P\}\ S\ \{Q\}$ is said to be stable if Q is stable.*

We thus can make the notion of validity (or correctness) of a specification precise. Validity is first defined with respect to a finite set of output variables, and extended thereafter.

Definition 3 (Validity) *Let* $\{P\}\ S\ \{Q\}$ *be a specification formula, and* V *a finite subset of Out. We say that this specification is* valid w.r.t. V*, which is denoted by*

$$\models_V \{P\}\ S\ \{Q\},$$

if for any $\sigma \in Init$ *such that* $\sigma \models P$*, for any sequence* $\sigma \longrightarrow^* \sigma'$ *such that* $V \subseteq supp(\sigma')$*, we have*

$$\sigma' \models Q.$$

More generally, we say that this specification is valid *which is denoted by* $\models \{P\}\ S\ \{Q\}$*, if for any* V*, finite subset of Out,* $\{P\}\ S\ \{Q\}$ *is valid w.r.t.* V*.*

3.3 Provability of a specification

We now want to introduce the notion of *provability* for a specification. Due to the iterative and non-deterministic nature of the operational semantics, the proof of a specification is established by producing an invariant. The notion of invariant defined here is relative to a program, to a subset of initial states and to a subset of output variables.

Definition 4 (Invariant) *A formula* I *is an* invariant *for the program* S*, the precondition* P *and the finite set* $V \subseteq Out$ *iff*

$$\textit{for all } \sigma \textit{ such that } \exists\ \sigma_0 \in Init \textit{ with } \sigma_0 \models P \textit{ and } \sigma_0 \underset{V}{\Longrightarrow}^* \sigma,\ \textit{we have } \sigma \models I,$$

where $\underset{V}{\Longrightarrow}$ *denotes a demand-driven transition relative to program* S *and subset* V*.*

Definition 5 (Provability) *Specification* $\{P\}\ S\ \{Q\}$ *is said to be* provable *w.r.t. a finite subset* V *of Out which is denoted by*

$$\vdash_V \{P\}\ S\ \{Q\},$$

if there exists an invariant I *w.r.t.* S*,* P *and* V *such that*

$$\textit{for all } \sigma \textit{ such that } supp(\sigma) \supseteq V, \qquad (\sigma \models I) \Rightarrow (\sigma \models Q).$$

More generally, a specification $\{P\}\ S\ \{Q\}$ *is said to be* provable *if it is provable for any finite subset* V *of Out.*

For instance, *true* is a trivial invariant for any specification, but it is of no use to establish provability in the general case, since we will not be able to prove that *true* $\Rightarrow Q$ for any Q.

In the case of our convolution example, an invariant could be

$$\begin{array}{ll} & \forall i \geq 1 : \Delta(x[i]) \\ \wedge & \forall j \in \{1 \ldots 4\} : \Delta(a[j]) \\ \wedge & \forall j \in \{1 \ldots 4\} : (Y[i_0, j] \triangleq Y[i_0, j-1] + a[j] * x[i_0 - j + 1]) \\ \wedge & (y[i_0] \triangleq Y[i_0, 4] \end{array}$$

We will see in the next section that this invariant is in some sense the canonical (strongest) invariant for this program.

In [2] we have proved that this notion of provability is *sound*, i.e. that every provable specification is valid. The converse problem (is any valid specification provable?) is briefly discussed below.

4 Towards a proof methodology

Proving a specification simply boils down to exhibiting an invariant. The only difficulty remains in proving that a given formula is an invariant for a given program. A handwaving method is to check that this formula remains true during each transition step of the program. In this section, we give a first attempt to develop more interesting proof methodologies.

4.1 Canonical invariants

The canonical invariant of an ALPHA program is the translation of the program in our logical language. It contains the whole semantics of the program, i.e. all the computations of its variables, including dependence relations. It expresses that

- each input variable $A[\vec{i}]$ is defined: $\forall \vec{i} \in \mathcal{D}_A : \Delta(A[\vec{i}])$;
- if a non-input variable $X[\vec{i}]$ is defined, then the variables it is depending on are defined, and its value is equal to the value of the defining expression: $\forall \vec{i} \in \mathcal{D}_X^p : X[\vec{i}] \triangleq \varphi(Y[f(\vec{i})], Z[g(\vec{i})])$.

Instead of reasoning on the program's text and doing substitutions, we will be able to reason directly on the logical expression of the program. The canonical invariant is formally defined by the following.

Definition 6 (Canonical invariant) *Let S be a program and P an assertion.*
The canonical invariant *I_C of the pair (P, S) is defined as the conjunction of*

$$P,$$

of all formulas

$$\forall \vec{i} \in \mathcal{D}_A : \Delta(A[\vec{i}])$$

for each input data-field A, and of all formulas

$$\forall \vec{i} \in \mathcal{D}_X^p : X[\vec{i}] \triangleq \varphi(Y[f(\vec{i})], Z[g(\vec{i})])$$

for each non-input data-field X.

It is easy to prove that I_C is indeed an invariant w.r.t. P, S, V in the sense of Definition 4: we start from an initial state where only input variables are defined, and each transition computes a variable according to its definition expression. The canonical invariant is the largest invariant we can produce for a program, since it contains all the semantics of this program. It is the key to establish the completeness of our proof method: since we are able to construct this invariant for any program, we are able to show that every valid specification is provable, under some restrictions on the form of the program and on the postcondition. We must assume that each output variable only depends on a finite number of variables. This restriction eliminates the case where we cannot reach a final state because of an infinite branch in the dependence graph. Moreover, we have to assume that the postcondition contains only program's variables. Otherwise we would not be able to prove that the canonical invariant, which contains only program's variables, implies the postcondition.These restrictions are "reasonable" in the sense that they concern all "useful" programs. With these restrictions, we show that, if the specification is valid, a final state satisfying the canonical invariant also satisfies the postcondition. This completeness result is formally proved in [2]. We don't discuss here any more this theoretical point.

4.2 Proving partial properties

Even though canonical invariants contain too much information, they will be useful to prove partial properties, i.e. properties that do not need a complete knowledge of the variables' values. Let us remark that if I is invariant for some program S, and if I' is a formula such that $I \Rightarrow I'$, then I' is also an invariant for S. As we know that the canonical invariant is an invariant, it suffices in most cases to prove that the formula we want to establish is implied by the canonical invariant of the considered program.

Let us go back to our convolution example of section 1. In this example, we have

$$\begin{aligned} In &= \{a[j] \mid 1 \leq j \leq 4\} \cup \{x[i] \mid 1 \leq i\} \cup \{Y[i,0] \mid 1 \leq i\} \\ Out &= \{y[i] \mid 4 \leq i\} \end{aligned}$$

Let us fix a value $i_0 \geq 4$ for i, and let $|.|$ denote the absolute value function. A specification could be

$$\begin{aligned} P &\equiv \forall j \in \{1 \ldots 4\} : |a[j]| \leq M \wedge |x[i_0 + j - 1]| \leq K \\ Q &\equiv |y[i_0]| \leq 4KM \end{aligned}$$

Taking $V = \{y[i_0]\}$, we can easily see that $\{P\}\ S\ \{Q\}$ is valid w.r.t. V.

The canonical invariant I_C of this program for this specification and this subset V is

$$\begin{aligned} & \forall i \geq 4 : \Delta(x[i]) \ \wedge \ \forall j \in \{1 \ldots 4\} : \Delta(a[j]) \\ \wedge \quad & \forall i \geq 4 : \forall j \in \{1 \ldots 4\} : (Y[i,j] \triangleq Y[i,j-1] + a[j] * x[i_0 - j + 1]) \\ \wedge \quad & \forall i \geq 4 : y[i] \triangleq Y[i,4] \end{aligned}$$

Let now I be the following formula

$$\begin{aligned} & \forall j \in \{1 \ldots 4\} : \Delta(Y[i_0,j]) \Rightarrow (|Y[i_0,j]| \leq jKM) \\ \wedge \quad & y[i_0] \triangleq Y[i_0,4] \end{aligned}$$

It is easy to show by induction on j that $I_C \Rightarrow I$. Thus I is an acceptable invariant. Moreover, this invariant implies the desired postcondition for final states.

On this simple example, the proof could naturally have been done "by hand". But reasoning on invariants helps us to respect a logical framework that guarantees the validity of the result and provides a methodology for more complex specifications.

4.3 Unambiguous invariants

The canonical invariant of a program is naturally easy to produce, but it may be too complex for our purpose. We may want to have an invariant that contains only "useful" informations. For instance, we may be interested in the final values of the output variables, but not in the way they were computed. An *unambiguous* invariant is an invariant that contains enough information to describe the final values of the output variables of the considered program. In some sense, it captures all the semantics of the program except the dependence relations. More formally, we have the following definition.

Definition 7 (Unambiguous invariant) *Let I be an invariant w.r.t. S,P,V. We say that I is* unambiguous *with respect to some finite subset V of Out if, for any pair of states σ and σ' both final with respect to V, such that I is true in σ and in σ', and that σ and σ' are equivalent modulo In, then σ and σ' are equivalent modulo V.*

A canonical invariant is unambiguous.

4.4 Proving equivalence properties

The notions of canonical and unambiguous invariants are particularly well-adapted to prove that two programs are equivalent. Let us define more precisely this notion of equivalence.

Definition 8 (Equivalence between programs) *If P is an assertion on input variables and V a finite subset of Out, we say that two programs are* equivalent from P with respect to V *if they have the same input and output variables, anf if, starting from the same initial state satisfying P, they yield final states (w.r.t. V) that are equivalent modulo V.*

When P is true, we simply say that the programs are equivalent w.r.t. V.

Intuitively, the notion of unambiguous invariant means that such an invariant contains enough information to describe the results of the program. As expressed by the following fact, two programs respecting the same unambiguous invariant are equivalent.

Proposition 1 *Let S and S' be two programs having the same input and output sets In and Out, let P be an assertion on input variables, and let V be a finite subset of Out.*

If S and S' admit a common unambiguous invariant I w.r.t. P and V, then S and S' are equivalent from P w.r.t. V.

The proof of this proposition is straightforward, by definition of an unambiguous invariant.

As above, instead of doing handwaving computations on transitions, checking that two programs respect the same unambiguous invariant can be done by checking logical implications between this invariant and the respective canonical invariants. To prove equivalence, we may

- either prove that there is a logical implication between the two canonical invariants
- or find a simpler unambiguous invariant, and prove that it is implied by both canonical invariants,

or any combination of these two strategies using intermediate unambiguous invariants.

As an example, let us consider the retiming problem of section 1. Let I_1 be the canonical invariant of the first program with respect to subset $V = \{X[i] \mid 0 \leq i \leq N+1\}$ and to precondition *true*. We have (X_1' stands for `X1prim`)

$$\begin{aligned} I_1 \quad \equiv \quad & \forall t \in \{0 \ldots N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge \quad & \forall t \geq 0 : \Delta(C[t]) \\ \wedge \quad & \forall t \in \{0 \ldots N+1\} : X_1'[t] \triangleq 2 * X_1[t] \\ \wedge \quad & \forall t \in \{1 \ldots N+1\} : X_2'[t] \triangleq 2 * X_2[t] \\ \wedge \quad & \forall t \in \{1 \ldots N\} : X[t] \triangleq (C[t] \ ? \ X_1'[t] : X_2'[t]) \\ \wedge \quad & X[0] \triangleq (C[0] \ ? \ X_1'[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1] \ ? \ 0 : X_2'[N+1]) \end{aligned}$$

Let I_2 be the canonical invariant of the second program. We have

$$\begin{aligned} I_2 \quad \equiv \quad & \forall t \in \{0 \ldots N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge \quad & \forall t \geq 0 : \Delta(C[t]) \\ \wedge \quad & \forall t \in \{0 \ldots N+1\} : X[t] \triangleq 2 * X'[t] \\ \wedge \quad & \forall t \in \{1 \ldots N\} : X'[t] \triangleq (C[t] \ ? \ X_1[t] : X_2[t-1]) \\ \wedge \quad & X'[0] \triangleq (C[0] \ ? \ X_1[0] : 0) \\ \wedge \quad & X'[N+1] \triangleq (C[N+1] \ ? \ 0 : X_2[N]) \end{aligned}$$

Let now I be the following formula.

$$\begin{aligned} I \quad \equiv \quad & \forall t \in \{1 \ldots N\} : X[t] \triangleq (C[t] \ ? \ 2 * X_1[t] : 2 * X_2[t-1]) \\ \wedge \quad & X[0] \triangleq (C[0] \ ? \ 2 * X_1[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1] \ ? \ 0 : 2 * X_2[N]) \end{aligned}$$

By a case analysis on the values of t, it is easy to show that

$$I_1 \Rightarrow I \text{ and } I_2 \Rightarrow I.$$

Thus I is an invariant. Moreover, as I precisely defines all values of the output data-field X, I is unambiguous. We can conclude that the two programs are equivalent on their output.

Now, if we take as precondition

$$P \equiv \forall t \geq 0 : C[2*t] = \mathit{true} \wedge C[2*t+1] = \mathit{false}$$

and if we focus on the first program, its canonical invariant is modified into

$$\begin{aligned} I_1 \quad \equiv \quad & \forall t \geq 0 : C[2*t] = \mathit{true} \wedge C[2*t+1] = \mathit{false} \\ \wedge \quad & \forall t \in \{0 \ldots N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge \quad & \forall t \in \{0 \ldots N+1\} : X_1'[t] \triangleq 2 * X_1[t] \\ \wedge \quad & \forall t \in \{1 \ldots N+1\} : X_2'[t] \triangleq 2 * X_2[t] \\ \wedge \quad & \forall t \in \{1 \ldots N\} : X[t] \triangleq (C[t] \ ? \ X_1'[t] : X_2'[t]) \\ \wedge \quad & X[0] \triangleq (C[0] \ ? \ X_1'[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1] \ ? \ 0 : X_2'[N+1]) \end{aligned}$$

to take into account the precondition. Notice that, since we use "real" equalities between $C[t]$ and either *true* or *false*, this formula implicitly expresses the fact that $C[t]$ is defined.

As above, we have

$$I_1 \Rightarrow I,$$

where I is modified the same way:

$$\begin{aligned} I \quad \equiv \quad & \forall t \geq 0 : C[2*t] = \mathit{true} \wedge C[2*t+1] = \mathit{false} \\ \wedge \quad & X[0] \triangleq (C[0] \ ? \ 2*X_1[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1] \ ? \ 0 : 2*X_2[N]) \\ \wedge \quad & \forall t \in \{1 \ldots N\} : X[t] \triangleq (C[t] \ ? \ 2*X_1[t] : 2*X_2[t-1]) \end{aligned}$$

It is finally easy to show that $I \Rightarrow I'$, with

$$\begin{aligned} I' \quad \equiv \quad & \forall t \in \{0 \ldots \lfloor N/2 \rfloor\} : X[2*t] \triangleq 2*X_1[2*t] \\ & \wedge X[2*t+1] \triangleq 2*X_2[2*t] \end{aligned}$$

5 Discussion

Our contribution is to provide the ALPHA language with an external logical framework to prove functional properties of programs. The key idea is to associate to each program a logical formula, which captures its functional behavior without ambiguity. It is called its *canonical invariant*. This formula can be seen as a kind of "weakest invariant" for the program. Proving a property which does not depend on the operational scheduling of the program amounts thus to checking that it is logically implied by this invariant.

Because of the very simple structure of the ALPHA programs, we can give an explicit form for the stronest invariant. The key feature is that the ALPHA variables are assigned only once: they can thus be considered as logical variables as well, provided one initially sets them to some undefined special value $\perp$. As expected, this invariant is essentially the same as the program text, up to the semantics of the "=" symbol. This is the reason why the refinement calculus on ALPHA program is so rich: most legal manipulations on logical predicates can be carried out directly at the level of programs.

We have moreover exhibited a number of sufficient conditions for a predicate to be an unambiguous invariant of an ALPHA program, if not its weakest canonical one. In some cases, for instance of the property under study refers to only a subset of the program variables, it may thus be possible to use a simpler invariant: simple properties can be proved by simpler proofs. Also, we have shown that usual methods for the equivalence proof of ALPHA programs are in fact special cases of such use of invariants. Again, the syntactic identity between a program and its canonical invariant is the key. This validates many "handwaving" manipulations found in the literature.

This preliminary work needs to be extended into several directions.

- Explore the equivalence proofs of ALPHA programs as found in the literature, and recast them into this logical framework.
- Understand more precisely how simple properties can be proved by simpler proofs. For instance, if a property is only concerned with a subset of the program variables, define a kind of "partially unambiguous invariant" which refers only to them.
- Understand how the logical framework sketched in this paper leads to revisit the refinement techniques for ALPHA programs. For instance, it would be interesting to exploit the complementarity between both approaches, internal and external.
- Extend this framework to modular ALPHA programs, as defined in [3].

Acknowledgments It is a pleasure to acknowledge many fruitful encouragements and discussions with Patrice Quinton and Sanjay Rajopadhye on this work.

References

[1] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Text and Monographs in Computer Science. Springer Verlag, 1991.

[2] L. Bougé and D. Cachera. A logical framework to prove properties of Alpha programs. Research Report 3031, INRIA, Grenoble, France, November 1996.

[3] F. Dupont de Dinechin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. PhD thesis, Univ. Rennes I, France, January 1997.

[4] C. Dezan and P. Quinton. Verification of regular architectures using Alpha: a case study. Internal publication 823, IRISA, Campus de Beaulieu, Rennes, France, 1994.

[5] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–583, 1969.

[6] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[7] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques*. PhD thesis, Univ. Rennes I, France, December 1989.

[8] L. Nédelka. Étude expérimentale des systèmes de transitions des programmes Alpha. Master's thesis, IRISA, Campus de Beaulieu, Rennes, France, 1995.

[9] S.V. Rajopadhye. An improved systolic algorithm for the algebraic path problem. *INTEGRATION: The VLSI Journal*, 14(3):279–296, Feb 1993.

[10] D.K. Wilde. The Alpha language. Technical Report 999, IRISA, Campus de Beaulieu, Rennes, France, January 1994.

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)
ISSN 0249-6399