1400226285
Còpia 1

Automatic Verification of Programs: Algorithm ALICE

V.R. Palasí Lallana

Report LSI–96–34–R

Automatic verification of programs: algorithm ALICE*

V.R. Palasí Lallana
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
e-mail: vicent@goliat.upc.es

Abstract

This paper aims to introduce a method for verification of programs, which is fully automatic. This method consists in an algorithm called ALICE which, given a program and an algebraic specification, answers if the program is correct w.r.t. the algebraic specification. No user intervention is needed (except, of course, the writing of the program and the specification). The paper also proves that the problem of determining the correctness of a program w.r.t. an algebraic specification is undecidable (even if only partial correctness is required).

1 Introduction

Since 1969, when software's chronic crisis was detected, it has been clear that the lack of an automatic procedure of verification (that is, an automatic procedure for checking the correctness or incorrectness of a program at compile-time) is a great drawback (see, for instance, [Gib94]).

We say that a program is correct if the behaviour expected from it is the same as its actual behaviour. Therefore, it is obvious that we can refer to the correctness of a program only if we have a description of the behaviour expected from it, which will be called the specification. This specification must be written in a formal language if we want to study the problem of correctness from an automatic point of view.

Although a lot of research has been employed in order to find an automatic procedure which serves to obtain correct programs, the most interesting results can be divided into two groups. Firstly, there are the procedures derived from Hoare Logic ([Hoa71], [Gri81],

*This paper is a summary of several parts of V.R. Palasi's future thesis "Application of behavioral equivalence to automatic verification of programs".

[DiF88], [Coh90]). Secondly, there are the techniques known as transformational development ([Bal81], [Bau82], [PaS83]).

Though it is impossible to explain these theories here, we can say that none of them is fully automatic. In Hoare Logic, the user must supply the invariant suitable for each loop. In transformational development, the user must choose the transformation rule which will be applied in each moment. Consequently, these theories allow us to do "semiautomatic verification" or "computer-aided verification", but never "automatic verification".

This paper aims to introduce a method for verification of programs, which, unlike the aforementioned, is fully automatic. This method consists of an algorithm called ALICE [1] which, given a program and an algebraic specification, determines whether the program is correct w.r.t. the algebraic specification. No user intervention is needed (except, of course, the writing of the program and the specification).

The complete description of algorithm ALICE and the proof of its correctness appear in the following research reports: [Pal96a], [Pal96b], [Pal96c], [Pal96d]. However, the brevity of this paper makes it impossible to describe the algorithm with full particulars. Consequently, we explain the intuitive ideas and we shall only give a formal description when it is strictly necessary.

The structure of this paper is as follows. First, we describe the algebraic notation used in it (Section 2). Then, we explain a general idea of our method (Section 3). Later, we define the algebraic theory on which this method is based (Section 4). Sections 5 and 6 explain several parts of algorithm ALICE. Finally, some conclusions and future research lines are outlined (Section 7).

2 Algebraic notation of this paper

We begin by concisely describing the notation of this paper (for a more detailed description, see [Pal96a]). The reader should realize that many-sorted algebra with Horn clauses is used

Definition 1. A simple signature Σ is a tuple $\Sigma = (S, F)$ where the members of S are called sorts and those of F, function symbols or operations.

We refer to S as $sorts(\Sigma)$ and to F as $opns(\Sigma)$. Variables of sort s are referred to as $vars(s)$.

[1]It is the acronym of"ALgebraic Inference of the Correctness of Environments".

If an operation f has n arguments belonging to sorts $w_1, ..., w_n$, respectively, and its result belongs to sort s; we shall write $f : w_1 ... w_n \longrightarrow s$. We define $opns(\Sigma)_{\lambda,s}$=$\{f \mid f : \longrightarrow s\}$ and $opns(\Sigma)_{w1..wn,s}$=$\{f \mid f : w_1 ... w_n \longrightarrow s\}$.

Definition 2. Let there be two simple signatures $\Sigma_1 = (S_1, F_1)$ and $\Sigma_2 = (S_2, F_2)$. We shall write $\Sigma_1 \subseteq \Sigma_2$ if $S_1 \subseteq S_2$ and $F_1 \subseteq F_2$.

Definition 3. Let Σ = (S,F) be a simple signature and X a set of variables. We refer to the set of terms of sort s containing variables of X as $T_{\Sigma_s}(X)$. The set of terms (of any sort) with variables of X is called $T_\Sigma(X)$.

Furthermore, we define $T_\Sigma = T_\Sigma(\varnothing)$. The members of T_Σ are called *ground terms*.

Definition 4. Given a simple signature Σ, we refer to a Σ-equation of arity n (that is, Σ-equation with n conditions) as $e : c_1 = d_1 \ \&...\& \ c_n = d_n \Rightarrow t_1 = t_2$. If a Σ-algebra A satisfies an equation e, then we shall write $A \models e$.

Definition 5. A simple specification is a tuple $SP = (\Sigma, E)$ where Σ is a simple signature and E a set of equations. Given a simple specification SP, we refer to its initial algebra as T_{SP}

3 General idea

The general idea of algorithm ALICE appears in Figures 1 and 2. If we want to check the correctness of a program P w.r.t. an algebraic specification SP_1, we must see if the program and the specification are "equivalent" according to a reasonable notion of equivalence (which is defined in Subsection 6.1).

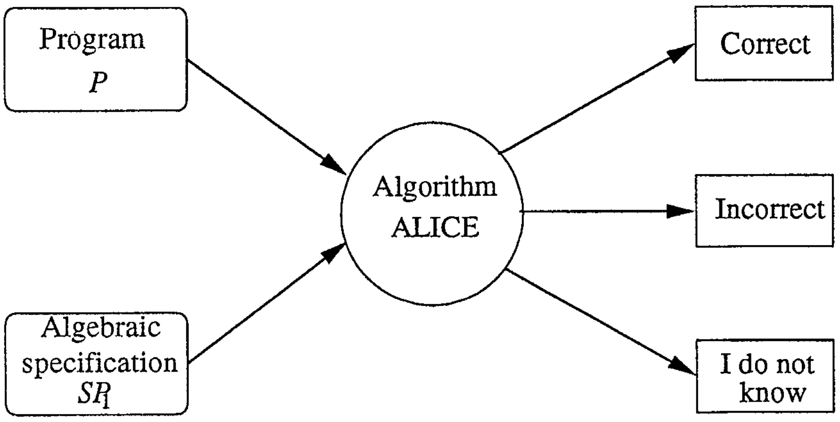

Figure 1: External view of algorithm ALICE.

Now, comparing a program and an algebraic specification is comparing two mathematical objects which are of a different kind. Consequently, it seems easier to compare two specifications. Therefore, what we will do is to transform program P into an algebraic specification SP_2 that is "equivalent" to it. This transformation will be made by algorithm α (which is studied in Section 5).

When algorithm α is finished, we will obtain two specifications SP_1 and SP_2. We will have to check if these specifications are equivalent (according to the meaning of equivalence defined in Section 4). Algorithm β creates an algebraic specification SP_3 and a set I of inductive theorems such that proving the equivalence between SP_1 and SP_2 is the same as proving I in the initial algebra of SP_3.

To sum up, the problem of determining whether P is correct w.r.t. algebraic specification SP_1 has been reduced to proving several inductive theorems in a given initial algebra. Therefore, we can use an inductive theorem prover (algorithm γ) in order to solve this problem.

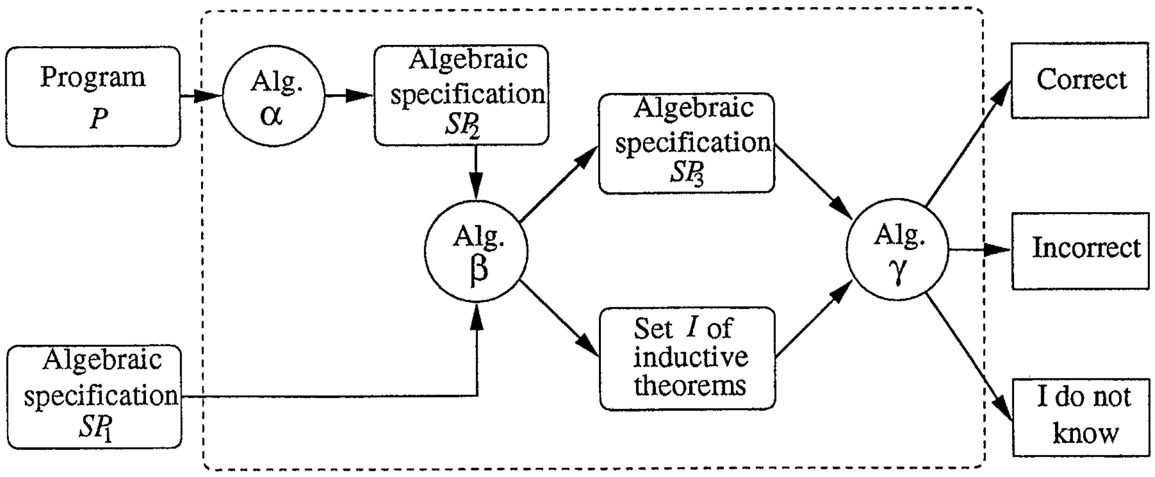
Program P
Alg. α
Algebraic specification SP2
Algebraic specification SP3
Alg. β
Alg. γ
Set I of inductive theorems
Algebraic specification SP1
Correct
Incorrect
I do not know

Figure 2: Internal structure of algorithm ALICE.

We should like to make two comments before finishing this section. The first one is that algorithm γ is just an inductive theorem prover. Since bibliography about proof of inductive theorems exists (see [KoR90], [Red90] for Horn clauses), this algorithm will not be explained here.

The second comment is that algorithm ALICE is not a decision procedure, since one of its parts is an inductive theorem prover. However, it can be proved (the proof is in appendix B) that the problem of determining whether a program is correct w.r.t. an algebraic specification is undecidable (even if only partial correctness is required) . Therefore, no decision procedure exists for this problem.

4 Visible semantics

Though there have been several proposals of initial ([GTW75], [GTW78]), final ([Wan79], [Kam83]) and behavioral ([Niv87], [Kna93], [BBK94]) algebraic semantics, none of them is completely suitable for dealing with the problem of proving the correctness of an algorithm w.r.t. an algebraic specification. The following example makes this clear.

4.1 Intuitive ideas

Let us suppose that there is an imperative programming language in which the operation of multiplication is not built-in, and also that a function to compute this operation is programmed. The result could be as follows[2]:

```
obs *(a,b:nat) ret c: nat
    c:=0;
    while b > 0 do
        c:=c+a;
        b:=b-1
    endwhile
endfunction
```

If we wish to study the correctness of this function, we must have to write a specification of it. The result can be as follows (initial semantics is used):

```
spec SP₁ is
    sorts
        nat
    signature
        *: nat nat ⟶ nat
    equations
        ∀ a,b:nat
        *(zero,b)=zero
        *(suc(a),b)=+(*(a,b),b)
endspec
```

We assume that operations suc, zero *and* + *(with prefix notation) have been defined. Their signatures and equations have been omitted for reasons of brevity.*

We shall consider that function "*" is correct if it is equivalent to specification SP_1 according to some reasonable notion of equivalence.

Following the general idea explained in Section 3, we transform function "*" into the following specification (via algorithm α):

[2]We use the programming language AL, which is described in Subsection 5.1. In this subsection the meaning of **obs** is also explained.

```
spec SP₂ is
    sorts
        nat
    signature
        *: nat nat ⟶ nat
    equations
        ∀ a,b:nat
        *(a,b)=eval_function("obs *(a,b:nat) ret c: nat ... endfunction",a,b)
endspec
```

Operation eval_function *is part of the algebraic semantics of the language. It is assumed that all the operations of this algebraic semantics have been defined. Their signatures and equations have been omitted for reasons of brevity. (For a more detailed description of the algebraic semantics of a language, see Section 5).*

Now, we can say that function "*" is correct w.r.t. specification SP_1 if specifications SP_1 and SP_2 (which is the algebraic semantics of function "*") are "equivalent". It is easy to see some properties that this equivalence must fulfil.

1. Two specifications can be equivalent despite having different signatures (in our example, SP_2 has at least one operation (eval_function) which does not belong to SP_1).

2. In fact, we can divide operations of a specification into two classes: those that we wish to specify (such as "*", in our example) and those that we describe only because they are required to define the former (such as "eval_function"). We call them "observable" and "hidden", respectively. It is easy to see that two equivalent specifications must have the same observable operations.

3. Finally, it is well known that a term represents a "computation" of the specified software system. Terms that have some hidden operation represent computation states that are not visible in the external behaviour of the system. Therefore, only terms that only have observable operations (hence, "totally observable terms") should be considered to define the equivalence.

Therefore, we can suppose that two specifications are equivalent if their totally observable terms "behave" in the same way. But this approach is not completely correct.

The problem with this approach becomes clear when we wish to specify data types like stacks, lists, sets, etc. For these data structures, it is advisable to obtain an abstraction of the internal implementation similar to that of the conventional behavioural semantics. To do this, the observable sorts[3] must be divided into two classes. The "nonvisible" ones are the sorts that have abstraction of the internal implementation (they will normally represent data structures). The remaining sorts are "visible" sorts (they will normally represent

[3]A sort s is observable if there is an observable operation of sort s

atomic types)[4]. Two specifications are equivalent if their visible results are the same, that is, if their totally observable terms of visible sort "behave" in the same way.

To sum up, two levels of "observability" are needed. That which distinguishes between observable and hidden operations and that which distinguishes between visible observable and nonvisible observable sorts.

Although there are several formulations of algebraic semantics (see [GTW78], [Kam83], [BBK94]) there is none which fulfils all the properties stated so far. One of the main problems is that algebraic semantics is traditionally defined between algebras that share the same signature and, as we have seen, this is not acceptable here.

Consequently, we must define an algebraic semantics suitable for solving our problem. This is done in the following subsection.

4.2 Formal definitions

In this subsection, we shall formally define an algebraic semantics (called "initial visible semantics") which fulfils all the properties needed for dealing with the problem of program correctness (that is, all the properties stated in the above subsection).[5]

Definition 6. A visible signature is a triple $(Vis, \Sigma_{Obs}, \Sigma_{All})$ such that $\Sigma_{Obs} = (S_{Obs}, F_{Obs})$, $\Sigma_{All} = (S, F)$ are simple signatures and, moreover, $Vis \subseteq S_{Obs}$ and $\Sigma_{Obs} \subseteq \Sigma_{All}$.

We refer to the tuple (Vis, Σ_{Obs}) as observable signature and, therefore, we define $sig\_obs(\Sigma) = (Vis, \Sigma_{Obs})$. The members of S_{Obs} and of F_{Obs} are called observable sorts and operations, respectively.

We call the members of Vis visible sorts. The members of $(S \backslash S_{Obs})$ and of $(F \backslash F_{Obs})$ are called hidden sorts and operations, respectively.

Definition 7. Suppose that $\Sigma = (Vis, \Sigma_{Obs}, \Sigma_{All})$ is a visible signature. We say that A is a Σ-algebra if it is a Σ_{All}-algebra.

We define T_Σ, $T_\Sigma(X)$ as $T_{\Sigma_{All}}$, $T_{\Sigma_{All}}(X)$, respectively. Analogously, for each sort $s \in sorts(\Sigma_{All})$, we define $T_{\Sigma_s}(X)$, T_{Σ_s} as $(T_{\Sigma_{All}})_s(X)$, $(T_{\Sigma_{All}})_s$, respectively.

A term is totally observable of visible sort if all its operations are observable and, moreover, its sort is visible.

[4]This issue is explained at great length in [Niv87] for conventional behavioural semantics.
[5]Visible semantics is completely defined in [Pal96a].

Definition 8. Let $\Sigma = (Vis, \Sigma_{Obs}, \Sigma_{All})$ be a visible signature. We define $TVis_\Sigma = \{t \mid t \in (T_{\Sigma_{Obs}})_s \wedge s \in Vis\}$.

Obviously, $TVis_\Sigma$ is a subset of T_Σ. The members of $TVis_\Sigma$ are called totally observable (ground) terms of visible sort.

A totally observable (ground) term of visible sort represents a computation result which is externally visible (that is, visible "from outside" the specified system). As was stated in the above subsection, the visible equivalence must take into account only the totally observable terms of visible sort. This can be seen in the following definition.

Definition 9[6]**.** Let Σ_A and Σ_B be two visible signatures such that $sig\_obs(\Sigma_A) = sig\_obs(\Sigma_B)$. Let A be a Σ_A-algebra and B a Σ_B-algebra. We say that A and B are visibly equivalent (and we write $A \equiv_V B$) if[7]:

$$\forall\, t_1, t_2 \in TVis_{\Sigma_A} \text{ it is fulfilled that } A \models t_1 = t_2 \text{ if and only if } B \models t_1 = t_2$$

Given the concept of the visible signature, that of the visible specification is easily definable.

Definition 10. A visible specification is a tuple $SP = (Vis, \Sigma_{Obs}, \Sigma_{All}, E)$, where $\Sigma = (Vis, \Sigma_{Obs}, \Sigma_{All})$ is a visible signature and E is a set of Σ-equations. We define $sig\_obs(SP) = (Vis, \Sigma_{Obs})$. We refer to the initial algebra of the simple specification (Σ_{All}, E) as T_{SP}.

We shall define the initial visible semantics of a specification. This semantics must have the expressive power of initial semantics and fulfil the principle that two visibly equivalent algebras should be considered equal according to initial visible semantics.

Definition 11. The initial visible semantics of a visible specification SP is defined as:

$$Vis - I[SP] = \{A \mid A \equiv_V T_{SP}\}$$

Definition 12. Let SP_1 and SP_2 be two visible specifications. We say that SP_1 and SP_2 are (initially) visibly equivalent if $Vis - I[SP_1] = Vis - I[SP_2]$.

5 Algorithm α

As has been said in Section 3, algorithm α is the algorithm which transforms an imperative program into an equivalent algebraic specification (see Figure 2).

[6]In [Pal96a] visible equivalence was defined starting from the concept of visible isomorphism, and Definition 9 was a lemma.

[7]Moreover, notice that $TVis_{\Sigma_A} = TVis_{\Sigma_B}$.

Now, we need a programming language in order to write a program. Obviously, algorithm α is different for each language, though the underlying ideas are always the same. In this section, we will explain these ideas and we will illustrate them with a particular example: algorithm α for AL programming language.

5.1 AL programming language

In this section, we make a description of AL language. We suggest reading it together with the AL program which is in appendix A.

AL (Adt language) is a language based on the concept of class. A class is a programming module and a user-defined type at the same time. That is to say, the concept of class in AL is the same as that of an object-oriented language, except that class inheritance is not defined in AL.

An AL program is a sequence of class declarations. The declaration of a class T has two parts. Firstly, under the reserved word **state**, type T is defined as a tuple of simpler types, which are called attributes of T (for instance, in a class "stack", type "stack" can be defined as a tuple of an array and a pointer).

Secondly, (under the reserved word **functions**), functions using type T are defined as conventional imperative functions. They have some parameters and one result and the possibility of declaring local variables and accessing the attributes of the type defined by the class.

In fact, in order to obtain "abstraction from implementation", only functions belonging to class T can directly access the attributes of type T[8].

Since each type is defined starting from simpler types, it is required that there are some built-in types. In AL, these are naturals, booleans and the type called **table** which allow us to define all kinds of data structures.

Each function or class can be defined in AL as observable (**obs**) or hidden (**hidden**). A hidden function or class can be used inside an AL program but is not visible from outside (it only serves for implementation). Obviously, all the functions belonging to a hidden class must be hidden too (the compiler checks this).

This is a short description of AL language. For a more complete and more formal description, see [Pal96c].

[8]However, it is easy to see that functions belonging to other classes can indirectly access them via functions of class T.

5.2 Problem of algebraic semantics of a language

We wish to define an algorithm which transforms a program into a visible algebraic specification that is "equivalent" to it (see Figure 2). But before describing this algorithm, we must explain what the word "equivalence" means for us.

We say that a program P and a visible algebraic specification SP_1 are equivalent if the semantics of P is the same as that of SP_1. Since the semantics of an algebraic specification has already been defined in subsection 4.2, we must only define that of a program. Moreover, since we have to compare the two semantics, it would also be useful to define the semantics of a program in an algebraic way.

In fact, there have been several attempts at defining the semantics of an imperative language via algebraic specifications (see for instance [Wan80] and [GoP81]). But these works cannot be applied here for the following reasons:

- They are oriented to the definition of the semantics of a language and not to that of a particular program.
- The way they define data types is not suitable here.

The example of subsection 4.1 can help us to see the last reason. Let us focus on the signature of operation "eval_function". Since function "*" has two parameters of natural type and one result of the same type, the signature of "eval_function" should be "function nat nat $\longrightarrow$ nat".

However, we should be able to translate any imperative function into algebraic notation. Therefore, there must be one operation such as "eval_function" for each combination of parameters and result types. The problem is greater in a programming language such as AL, which allows the user to define an indefinite number of new types. In fact, the number of operations such as "eval_function" would be infinite. Analogously, the number of operations such as "eval_assignment", "eval_expression", "eval_statement" (and, as a general rule, all the operations which deal with different types) would also be infinite.

This problem is solved in [GoP81] by creating the generic sorts *value* (which means any value regardless of its type) and *lval* (which means any list of *values*). A program (in AL an imperative function) is translated into the algebraic notation as an operation with the signature *lvalue* $\longrightarrow$ *value*. In this way, there is one single "eval_function", one single "eval_assignment" and so on.

But the example in Subsection 4.1 shows that this solution is not acceptable here. If we have a function declared as "**obs** $*$(a,b:**nat**) **ret** c: **nat**", its translation into algebraic notation must be an operation with the signature "nat nat $\longrightarrow$ nat", since it must be compared to an operation with the same signature (that defined by the user in SP_1).

Consequently, our solution must be type-dependent enough to translate each imperative function into an algebraic operation with the same types. But it must be type-independent enough to avoid having infinite "eval_function", "eval_assignment", etc.

5.3 The solution to this problem

The solution proposed here is dual. On the one hand, AL language will be specified from the sorts *value, lval*, which allows us to avoid the existence of infinite operations. On the other hand, we will add to each particular program a set of sorts and operations which cause each imperative function to be translated into an operation with the suitable signature.

In this way, the semantics of any program P will be a visible specification $Trad(P)$ composed of two parts.

1. A type-independent and program-independent part, which is not included here because it is similar to that of [GoP81]. Furthermore, it is fully described in [Pal96c].
2. A part that is different for each program and is composed of the following sorts and operations (which specify the data types of an AL program):
   - For each class T in the program, there are
     - One sort T called "sort associated to the class".
     - The following conversion operations:
       - $gen_T : T \longrightarrow value$
       - $esp_T : value \longrightarrow T$
     - The equation $esp_T(gen_T(v)) = v$ (where v is of sort *value*)
   - For each function with a heading as $< obskind >$ f($a_1 : T'_1,...,a_n : T'_n$) **return** $a_{n+1} : T'_{n+1}$ (where $< obskind >$ may be either **obs** or **hidden**) there are:
     - One operation $f : T_1 \times ... \times T_n \longrightarrow T_{n+1}$ called "operation associated to the function" (where, for any i, T_i is the sort associated to T'_i).
     - One equation $f(a_1, ..., a_n) = esp_{T_{n+1}}(eval - function(< obskind >$ f($a_1 : T'_1,...,a_n : T'_n$) **return** $a_{n+1} : T'_{n+1}, gen_{T_1}(a_1) :: gen_{T_2}(a_2) :: ... :: gen_{T_n}(a_n)))$[9].

Visible sorts are **nat** and **bool**. Observable ones are **nat**, **bool**, **table** and those associated to any class declared as "obs". Observable operations are the constructors of **nat**,

[9] a::b::c means the list of values [a,b,c].

obs i table and, also the operations associated to any function declared as "**obs**".[10]

To sum up, the semantics of P can be defined as:

$$Semantics(P) = Semantica(Trad(P))$$

Now, algorithm α is the algorithm which transforms a program into an algebraic specification that is equivalent to it (that is, a specification with the same semantics as the program). Therefore, algorithm α must be that which transforms P into $Trad(P)$[11].

Moreover, since we are dealing with initial visible semantics, the above statement is equivalent to:

$$Semantics(P) = Vis - I[Trad(P)]$$

And this equation is the fully formal definition of the semantics of any program P.

6 Algorithm β

6.1 The need for this algorithm

Algorithm ALICE aims to deduce whether program P is correct w.r.t visible specification SP_1 (that is to say, whether P and SP_1 are equivalent): in other words, whether P and SP_1 have the same semantics.

That is to say, algorithm ALICE aims to deduce whether:

$$Semantics(SP_1) = Semantics(P)$$

Since we are dealing with initial visible semantics, we have that:

$$Semantics(SP_1) = Vis - I[SP_1]$$

Furthermore, as we have seen in the above section:

$$Semantics(P) = Vis - I[Trad(P)]$$

Consequently, statement $Semantics(S) = Semantics(P)$ is equivalent to:

$$Vis - I[SP_1] = Vis - I[Trad(P)]$$

By definition 12, this statement is equivalent to:

[10]For a more detailed and more reasoned explanation of this, see [Pal96c].

[11]The definition of $Trad(P)$ shows that this is a very simple algorithm and its complexity is linear.

SP_1 and $Trad(P)$ are initially visibly equivalent

That is to say, the proof of the correctness of a program can be reduced to the proof of the initial visible equivalence between two algebraic specifications. As can be seen in Figure 2[12], the latter is implemented by the combination of algorithms β and γ. Now, we know that algorithm γ exists, since it is just an inductive theorem prover. Therefore, algorithm β is needed, because if this algorithm exists, we will have solved our problem.

6.2 Description of algorithm β

Algorithm β is the algorithm which reduces the proof of the equivalence between two specifications SP_1 and SP_2 in initial visible semantics to the deduction of inductive theorems, that is, of equalities in an initial algebra (see Figure 2).

If we were dealing with initial semantics, this reduction would be easy. In order to prove that SP_1 and SP_2 are equivalent, it would be enough to deduce satisfaction of equations belonging to SP_1 in the initial algebra of SP_2 and that of equations belonging to SP_2 in the initial algebra of SP_1.

The same method can be applied to the initial behavioural semantics, though it would be necessary to work with behavioural satisfaction instead of conventional satisfaction. But this is no problem, since the former can be reduced to the latter (see [BiH94], [BHW94]).

However, this method is not valid in initial visible semantics[13]. The reason is that SP_1 and SP_2 may have different signatures, and consequently the equations of SP_1 may not make sense in the initial algebra of SP_2 or vice versa.

Therefore, our method is different. We create a specification SP_3 whose initial algebra contains (at least) all the information contained in the initial algebras of SP_1 and SP_2.

Then, we reduce the proof of initial visible equivalence between SP_1 and SP_2 to the proof of a set I of inductive theorems on SP_3 (see Figure 2).

It is obvious that we must take care in the choice of SP_3 and the set I of inductive theorems. These issues are discussed in [Pal96d]. Here, we only explain the outcome which we have obtained.

Let us start by defining the concept of V-renaming. Its formal definition is not included here (the reader can find it in [Pal96d]). Informally, a specification SP_4 is a V-renaming of SP_2 if we can obtain SP_4 starting from SP_2 by changing all the names of the sorts and operations. Bijections $\theta : sorts(SP_2) \longrightarrow sorts(SP_4)$ and $\phi : opns(SP_2) \longrightarrow opns(SP_4)$

[12]In this figure, specification $Trad(P)$ is called SP_2.

[13]Although the concept of visible satisfaction can be defined (see [Pal96a]).

respectively mean the change of names of the sorts and the operations. They are called V-renaming bijections.[14]

Now, we will define the concept of V-reunion. Intuitively, a V-reunion of SP_1 and SP_2 is a simple specification which contains all the sorts and operations of SP_1 and the V-renaming of SP_2 and, also several other sorts and operations.

Concretely, the formal definition of a V-reunion is as follows.

Definition 13[15]. Let $SP_1 = (Vis, \Sigma_{Obs}, (\Sigma_{All})_1, E_1)$ and $SP_2 = (Vis, \Sigma_{Obs}, (\Sigma_{All})_2, E_2)$ be two visible specifications with the same observable signature.

Let there be a specification $SP_4 = (Vis_4, (\Sigma_{Obs})_4, (\Sigma_{All})_4, E_4)$, such that SP_4 is a V-Renaming of SP_2 via the V-renaming bijections θ and ϕ.

We say that a simple signature $SP_3 = (\Sigma_3, E_3)$ is a V-reunion of SP_1 and SP_2 via SP_4 if:

- $sorts(\Sigma_3) = sorts((\Sigma_{All})_1) \cup (sorts(\Sigma_{All})_4) \cup \gamma$ where $\gamma \notin (sorts((\Sigma_{All})_1) \cup sorts((\Sigma_{All})_4))$.
- $opns(\Sigma_3) = opns((\Sigma_{All})_1) \cup (opns(\Sigma_{All})_4) \cup F_{new}$ where F_{new} contains the following function symbols:
  - $yes : \longrightarrow \gamma$
  - $plus : \gamma \times \gamma \longrightarrow \gamma$
  - For any $s \in sorts(\Sigma_{Obs})$
    $trans_s : s \times \theta(s) \longrightarrow \gamma$

  where $yes, plus, trans \notin (opns((\Sigma_{All})_1) \cup opns((\Sigma_{All})_4))$
- $E_3 = E_1 \cup E_4 \cup E_{new}$ where E_{new} contains the following equations:
  - $plus(yes, yes) = yes$
  - $\forall\, s \in sorts(\Sigma_{Obs}) \quad \forall \sigma \in opns(\Sigma_{Obs})_{\lambda,s}$
    $trans_s(\sigma, \phi(\sigma)) = yes$
  - $\forall\, w1, .., wn, s \in sorts(\Sigma_{Obs}) \quad \forall \sigma \in opns(\Sigma_{Obs})_{w1..wn,s}$
    $trans_s(\sigma(t_1, t_2, ..., t_n), \phi(\sigma)(u_1, u_2, ..., u_n)) =$
    $plus(trans_{w_1}(t_1, u_1), plus(trans_{w_2}(t_2, u_2), ..., trans_{w_n}(t_n, u_n)...))$

[14]It is easy to see that the algorithm which creates a V-renaming of a visible specification has a linear complexity w.r.t. the input.

[15]In this definition, we have used the names γ, *yes*, *plus* and *trans*. There may be some trouble if, in SP_1 or SP_2, any of these names have already been used. This naming conflict is easily avoided by using names other than those belonging to SP_1 or SP_2.

where $E_{new} \cap (E_1 \cup E_4) = \varnothing$

Bijections θ and ϕ will be called V-reunion bijections. We can state the following theorem.

Theorem 14. Let $SP_1 = (Vis, \Sigma_{Obs}, (\Sigma_{All})_1, E_1)$ and $SP_2 = (Vis, \Sigma_{Obs}, (\Sigma_{All})_2, E_2)$ be two visible specifications with the same observable signature. Let SP_3 be a V-reunion of SP_1 and SP_2 via any specification, where the V-reunion bijections are called θ and ϕ.

Then, the following statements are equivalent:

- $\forall s \in Vis$, $\forall x_1, x_2 \in vars(s)$; $y_1, y_2 \in vars(\theta(s))$ it is fulfilled that $(T_{SP_3} \models trans_s(x_1, y_1) = yes \ \& \ trans_s(x_1, y_2) = yes \ \Rightarrow y_1 = y_2) \wedge$ $(T_{SP_3} \models trans_s(x_1, y_1) = yes \ \& \ trans_s(x_2, y_1) = yes \ \Rightarrow x_1 = x_2)$
- SP_1 and SP_2 are initially visibly equivalent.

Proof. The proof is in [Pal96d]. □

That is to say, proving the initial visible equivalence between two specifications has been reduced to proving several inductive theorems over specification SP_3. But this reduction is the aim of algorithm β

Therefore, algorithm β is the algorithm that, starting from two specifications, builds their V-reunion and the inductive theorems stated in Theorem 14. It is easy to see that the complexity of this algorithm is linear.

7 Conclusions and future work

An algorithm (called ALICE) which automatically deduces whether a program is correct w.r.t. an algebraic specification has been explained. In fact, the algorithm reduces the proof of correctness to the proof of some inductive theorems. This latter can be implemented by an inductive theorem prover, such as those described in the bibliography.

Though ALICE serves for performing automatic verification of programs, it is not a decision procedure (in fact, no decision procedure exists for this problem, see [Pal96b]). Therefore, it is advisable to use it class by class. That is to say, ALICE is individually applied to each class. If the result is "correct", the class is correct. If the result is "incorrect", there is an error in the class and we will have to debug it. On the other hand, if the algorithm cannot give a conclusive answer, we will have to mark the class as "doubtful".[16]

[16]Because of the existence of these "doubtful" classes, it is not possible to completely dispense with the traditional testing at run-time.

There are two future research lines. On the theoretical side, we wish to extend this method of verification to object-oriented programming. The aim is to describe an algorithm such as ALICE for object-orientation and to prove its correctness. To do this, we will probably have to use order sorted algebra ([GoM92]) instead of many-sorted algebra.

On the practical side, we wish to build an environment based on algorithm ALICE, which allows us to develop a program in the following way. First, a prototype of the program in algebraic specification is built. Then, the programmer translates the algebraic notation into imperative code class by class. When each class is translated, algorithm ALICE is used in order to determine its correctness. Finally, we obtain an imperative program which is developed with a "seamless development" method ([Coa91]) and is partially or completely verified.[17].

8 References

[Bal81] BALZER, R. *Transformational Implementation: An Example.* IEEE Transactions on Software Engineering **SE-7**, num. 1 (January 1981), pp. 3-14.

[Bau82] BAUER, F.L. *From Specifications to Machine Code: Program Construction Through Formal Reasoning.* Proceedings of Sixth International Conference on Software Engineering, Washington, 1982, pp. 84-91.

[BBK94] BERNOT, G. BIDOIT, M. KNAPIK, T. *Behavioural approaches to algebraic specifications.* Acta Informatica **31** (1994), pp. 651-671.

[BeT87] BERGSTRA, J.A. TUCKER, J.V. *Algebraic Specifications of Computable and Semicomputable Data Types.* Theoretical Computer Science, 50 (1987), pp. 186-200.

[BiH94] BIDOIT, M. HENNICKER, R. *Proving Behavioural Theorems with Standard First-Order Logic.* Research Report LIENS 94-11. Laboratoire d'Informatique. École Normal Supèrieure (1994).

[BHW94] BIDOIT, M. HENNICKER, R. WIRSING, M. *Behavioural and Abstractor Specifications.* Research Report LIENS 94-10. Laboratoire d'Informatique. École Normal Supèrieure (1994).

[Coa91] COAD, P. *OOA & OOD: a continuum of representation.* Journal of Object-Oriented Programming **3**, num. 6 (February 1991), pp. 55-56.

[17]Though a class be already translated, it is useful to preserve its algebraic specification as documentation and as a run-time test of the "doubtful" classes. (This run-time test will consist in running the imperative code and the algebraic specification at the same time. The environment will generate an exception if the two results are not the same.)

[Coh90] COHEN, E. *Programming in the 1990's. An Introduction to the Calculation of Programs.* Springer-Verlag, 1990. ISBN: 3-540-97382-6.

[DiF88] DIJKSTRA, E.W. FEIJEN, W.H.J *A Method of Programming.* Addison-Wesley, 1988. ISBN: 0-201-17536-3.

[Gri81] GRIES, D. *The Science of Programming.* Springer-Verlag, 1981. ISBN: 3-540-90641-X.

[Gib94] GIBBS, W.W. *Software's chronic crisis.* Scientific American **271**, num. 3 (September 1994), pp. 72-81.

[GoM92] GOGUEN, J.A. MESEGUER,J. *Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism and Partial Operations.* Theoretical Computer Science **105**, num. 2 (November 1992), pp. 217-273.

[GoP81] GOGUEN, J.A. PARSAYE-GHOMI, K. *Algebraic Denotational Semantics Using Parameterized Abstract Modules.* Formalization of Programming Concepts, LNCS **107** (1981), pp. 292-309.

[GTW75] GOGUEN, J.A. THATCHER, J.W. WAGNER, E.G. WRIGHT, J.B. *Abstract data types as initial algebras and correctness of data representations.* Proc. ACM Conference on Computer Graphics, Pattern and Data Structure, New York (1975), pp. 89-93.

[GTW78] GOGUEN, J.A. THATCHER, J.W. WAGNER, E.G. *An initial algebra approach to the specification, correctness and implementations of abstract data types.*, in: R.T.Yeh, ed., Current Trends in Programming Methodology; IV Data Structuring (Prentice-Hall, Englewood Clifts, NJ, 1978), pp. 80-149.

[Hoa71] HOARE, C.A.R. *Proof of a program: Find.* Communications of the ACM **14**, num. 1 (1971), pp. 39-45.

[Kam83] KAMIN, S. *Final data types and their specification.* ACM Transactions on Programming Languages and Systems **5** (1983), pp. 97-123.

[Kna93] KNAPIK, T. *Spécifications algébriques observationnelles modulaires: une semantique fondée sur une relation de satisfaction observationnelle.* Thèse de l'Université de Paris-Sud, Orsay 1993.

[KoR90] KOUNALIS, E. RUSSINOWITCH,M. *Mechanizing Inductive Reasoning.* Proceedings of the AAAI Conference, Boston, 1990, pp. 240-245.

[Niv87] NIVELA, P. *Semántica de Comportamiento en Lenguajes de Especificación.* PhD thesis, directed by Fernando Orejas Valdés. Facultat d'Informàtica. Universitat Politècnica de Catalunya (1987).

[Pal96a] PALASÍ, V.R *Visible Semantics: An Algebraic Semantics for Automatic Verification of Algorithms.* Research Report LSI-96-26-R. Departament de Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya (1996).

[Pal96b] PALASÍ, V.R. *Automatic Verification of Programs: the algorithm ALICE.* Research Report LSI-96-31-R. Departament de Llenguatges i Sistemes Informàtics (1996).

[Pal96c] PALASÍ, V.R. *Semàntica Algebraica del Llenguatge AL.* Research Report LSI-96-47-R. Departament de Llenguatges i Sistemes Informàtics (1996).

[Pal96d] PALASÍ, V.R. *Deducció Automàtica de l'Equivalència Inicial Visible.* Research Report LSI-96-64-R. Departament de Llenguatges i Sistemes Informàtics (1996).

[Pal96e] PALASÍ, V.R. *Semàntica externa.* To appear as a research report of the Departament de Llenguatges i Sistemes Informàtics (1996).

[PaS83] PARTSCH, H. STEINBRÜGGER, R. *Program Transformation Systems.* ACM Computing Surveys **15**, num. 3 (September 1983), pp. 199-236.

[Red90] REDDY, U. *Term Rewriting Induction.* Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern, 1990, pp. 162-177.

[Wan79] WAND, M. *Final algebra semantics and data type extensions.* Journal of Computer and System Sciences **19** (1979), pp. 27-44.

[Wan80] WAND, M. *First-Order Identities as a Defining Language.* Acta Informatica, **14** (1980), pp. 337-357.

A An AL class

```
obs seq_nat is (** Sequence of naturals (simplified for reasons of brevity)**)
    state
        members : table;
        length : nat;
    functions
        obs empty_seq return s:seq_nat is
          body
            s.length:=0
          endfunction;
          obs add (s:seq_nat; n:nat) return s2:seq_nat is
               i:nat;
            body
               i:=1;
               while i<=s.length do
                    s2.members[i]:=s.members[i]
               endwhile;
               s2.members[s.length+1]:=n;
               s2.length:=s.length+1;
          endfunction;
          obs member (s:seq_nat; i:nat) return n:nat is
            body
               n:=s.members[i]
          endfunction;
endclass.
```

B Software correctness is undecidable

In this appendix, we wish to prove that the problem of determining whether a program is correct w.r.t. an algebraic specification is undecidable (even if only partial correctness is required). The reasonings which we use can be applied to formal specification systems other than algebraic specification, as we will comment below.

To make the proof easier to write, we will suppose that an object-oriented programming language or a programming language based on abstract data types (such as AL[18]) is used. The algebraic semantics used in this proof is based on initial semantics, but it must allow us to define hidden symbols (such as visible semantics, which is explained in Section 4, or external semantics, which is described in [Pal96e]). This is necessary, since, if we cannot use hidden symbols, we cannot specify all semicomputable data types ([BeT87]).

B.1 Total correctness.

Firstly, we consider total correctness. In order to prove that the problem of checking total correctness is undecidable, we will reduce the halting problem to this. Let us suppose that we wish to determine whether a function P (with heading "**function** $P(p1 : t1;\ ...;\ pn : tn)$ **ret** $r : tr$") halts with a certain input $a1, ..., an$. To determine this, we build a function *Phalting* defined as follows:

function *Phalting* **ret** r:bool **is**

[18]In fact, in order to unify our notation, all examples in this appendix are written in AL language.

```
    temp:tr
    body
        temp:=P(a1,..,an);
        r:=true
endfunction
```

It is easy to see that P halts with the input $a1, ..., an$ if and only if *Phalting* returns "true".

Now, let us define the following algebraic specification:

```
spec HALTING_SPEC is
    sorts
        bool
    signature
        true: ⟶ bool
        Phalting: ⟶ bool
    equations
        Phalting=true
endspec
```

Function *Phalting* is (totally) correct w.r.t. algebraic specification HALTING_SPEC if and only if *Phalting* returns *true*. And *Phalting* returns true if and only if P halts. That is to say, we have reduced the checking of the halting of P to the checking of the total correctness of *Phalting* w.r.t. HALTING_SPEC. Since the halting problem is undecidable, so is the total correctness problem. (The reader should observe that this reasoning is not only applicable to algebraic specification, but to any kind of formal specification).

B.2 Partial correctness

Now, let us prove that checking partial correctness is also undecidable. Let us suppose a function P (with heading "**function** $P(p1 : t1; ...; pn : tn)$ **ret** $r : tr$"). We wish to determine whether this function halts with the input $a1, ..., an$. Let us build function *Phalting* as in the above subsection. As above, we have that P halts if and only if *Phalting* returns true.

Now, let us build algorithmically an algebraic specification (called PHALTING_SPEC) which has the same semantics as function *Phalting*. A way to obtain this is to apply algorithm α to function *Phalting* (that is, PHALTING_SPEC would be *Trad(Phalting)*, see Section 5).

It is easy to see that the semantics of PHALTING_SPEC only has two terms: *Phalting* i *true*, because all the other sorts and operations (those one which describe the language) are hidden. In fact, there are only two possibilities in the semantics of PHALTING_SPEC:

1. *Phalting* and *true* are equivalent.

2. *Phalting* and *true* are not equivalent.

We have that P halts if and only if *Phalting* returns true. Since the semantics of PHALTING_SPEC is the same as that of function *Phalting*, this means that P halts if and only if *Phalting* and *true* are equivalent in the semantics of PHALTING_SPEC (that is, if the first possibility occurs).

Therefore, P halts if and only if PHALTING_SPEC is equivalent to the following function:

```
function Phalting ret r:bool is
    body
       r:=true
endfunction
```

Now, in order to determine the equivalence between PHALTING_SPEC and this function, it is sufficient to have an algorithm which deduces partial correcteness. This is because, since the function always halts, proving its partial correctness is the same as proving its total correctness.

Therefore, we have reduced the halting problem to the problem of determining the partial correctness of a function w.r.t. an algebraic specification. Consequently, the latter is undecidable, which is what we wished to prove.

This reasoning can be extended to all formal specification systems S which fulfil that there is an algorithm such that, starting from a program P, it builds an S specification with the same semantics as P.

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1996

LSI-96-1-R	"(Pure) Logic out of Probability", Ton Sales.
LSI-96-2-R	"Automatic Generation of Multiresolution Boundary Representations", C. Andújar, D. Ayala, P. Brunet, R. Joan-Arinyo, and J. Solé.
LSI-96-3-R	"A Frame-Dependent Oracle for Linear Hierarchical Radiosity: A Step towards Frame-to-Frame Coherent Radiosity", Ignacio Martin, Dani Tost, and Xavier Pueyo.
LSI-96-4-R	"Skip-Trees, an Alternative Data Structure to Skip-Lists in a Concurrent Approach", Xavier Messeguer.
LSI-96-5-R	"Change of Belief in SKL Model Frames (Automatization Based on Analytic Tableaux)", Matías Alvarado and Gustavo Núñez.
LSI-96-6-R	"Compressibility of Infinite Binary Sequences", José L. Balcázar, Ricard Gavaldà, and Montserrat Hermo.
LSI-96-7-R	"A Proposal for Word Sense Disambiguation using Conceptual Distance", Eneko Agirre and German Rigau.
LSI-96-8-R	"Word Sense Disambiguation Using Conceptual Density", Eneko Agirre and German Rigau.
LSI-96-9-R	"Towards Learning a Constraint Grammar from Annotated Corpora Using Decision Trees", Lluis Màrquez and Horacio Rodríguez.
LSI-96-10-R	"POS Tagging Using Relaxation Labelling", Lluís Padró..
LSI-96-11-R	"Hybrid Techniques for Training HMM Part-of-Speech Taggers", Ted Briscoe, Greg Grefenstette, Lluís Padró, and Iskander Serail.
LSI-96-12-R	"Using Bidirectional Chart Parsing for Corpus Analysis", A. Ageno and H. Rodríguez.
LSI-96-13-R	"Limited Logical Belief Analysis", Antonio Moreno.
LSI-96-14-R	"Logic as General Rationality: A Survey", Ton Sales.
LSI-96-15-R	"A Syntactic Characterization of Bounded-Rank Decision Trees in Terms of Decision Lists", Nicola Galesi.
LSI-96-16-R	"Algebraic Transformation of Unary Partial Algebras I: Double-Pushout Approach", P. Burmeister, F. Rosselló, J. Torrens, and G. Valiente.

LSI-96-17-R	"Rewriting in Categories of Spans", Miquel Monserrat, Francesc Rosselló, Joan Torrens, and Gabriel Valiente.
LSI-96-18-R	"Strong Law for the Depth of Circuits", Tatsuie Tsukiji and Fatos Xhafa.
LSI-96-19-R	"Learning Causal Networks from Data", Ramon Sangüesa i Solé.
LSI-96-20-R	"Boundary Generation from Voxel-based Volume Representations", R. Joan-Arinyo and J. Solé.
LSI-96-21-R	"Exact Learning of Subclasses of CDNF Formulas with Membership Queries", Carlos Domingo.
LSI-96-22-R	"Modeling the Thermal Behavior of Biosphere 2 in a Non-Controlled Environment Using Bond Graphs", Angela Nebot, François E. Cellier, and Francisco Mugica.
LSI-96-23-R	"Obtaining Synchronization-Free Code with Maximum Parallelism", Ricard Gavaldá, Eduard Ayguadé, and Jordi Torres.
LSI-96-24-R	"Memoisation of Categorial Proof Nets: Parallelism in Categorial Processing", Glyn Morrill.
LSI-96-25-R	"Decision Trees Have Approximate Fingerprints", Víctor Lavín and Vijay Raghavan.
LSI-96-26-R	"Visible Semantics: An Algebraic Semantics for Automatic Verification of Algorithms", Vicent-Ramon Palasí Lallana.
LSI-96-27-R	"Massively Parallel and Distributed Dictionaries on AVL and Brother Trees", Joaquim Gabarró and Xavier Messeguer.
LSI-96-28-R	"A Maple package for semidefinite programming", Fatos Xhafa and Gonzalo Navarro.
LSI-96-29-R	"Bounding the expected length of longest common subsequences and forests", Ricardo A. Baeza-Yates, Ricard Gavaldà, and Gonzalo Navarro.
LSI-96-30-R	"Parallel Computation: Models and Complexity Issues", Raymond Greenlaw and H. James Hoover.
LSI-96-31-R	"ParaDict, a Data Parallel Library for Dictionaries (Extended Abstract)", Joaquim Gabarró and Jordi Petit i Silvestre.
LSI-96-32-R	"Neural Networks as Pattern Recognition Systems", Lourdes Calderón.
LSI-96-33-R	"Semàntica externa: una variant interessant de la semàntica de comportament", (written in Catalan) Vicent-Ramon Palasí Lallana.
LSI-96-34-R	"Automatic verification of programs: algorithm ALICE", V.R. Palasí Lallana.