ADA REUSABLE COMPONENTS FOR THE CONSTRUCTION OF ROBUST PROGRAMS BY MEANS OF ALGEBRAIC SPECIFICATIONS OF ABSTRACT DATA TYPES *

Francis LOSAVIO
Centro de Ingeniería de Software Y Sistemas ISYS
Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela.
Ap. 47002, Los Chaguaramos 1041-A, Caracas, Venezuela. Fax: 58-2-662-71-21
e-mail: flosavio@dino.conicit.ve

ABSTRACT. Algebraic specifications of abstract data types, taking account of genericity and exception handling, are used here as a guideline to derive reusable Ada components. This derivation is facilitated by an automatic tool which builds Ada package skeletons from the corresponding algebraic specification. The development of the Ada code considers two different levels of abstraction: the algebraic specification level, involving a specification language and the implementation level, involving the Ada language. We justify the translation of the algebraic specifications, expressed in the Pluss language, into valid Ada constructs. The fact of considering parameterized algebraic specification of abstract data types and exception handling features assure the construction of robust and reusable components, independently from the implementation language used. The major goal of this work is to show that algebraic specifications may be practically used in software development, within an assisted program construction context.

Key words: abstract data types, algebraic specifications, Ada resuable components, robust programs

1. INTRODUCTION.

The work presented in this paper is part of a major research project on systematic software construction undertaken by a work team at the Software Engineering Research Center, ISYS, Caracas, and it is aimed to study systematic application of an algebraic specification formalism, taking account of genericity and exceptional situations, for practical construction of robust and reliable software. Ada code [8] is interactively developed, within an assisted program construction framework [14], [13], from the algebraic specification of an abstract data type taking account of genericity and exception handling features. In this context, an *Ada component* is a generic package constructed from the corresponding algebraic specification. The specification language used is Pluss [10], [7], [3]. The axioms at specification levels are used by the implementor as

* This research is partially supported by the Consejo de Desarrollo Científico y Humanístico (CDCH) of the U.C.V. and the french Postgraduated Cooperation Program in Informatics of the CEFI-CONICIT.

guidelines for the implementation of the package body part and they are not directly involved in the process. The resulting Ada components are robust and generic units since we are using a formalism which takes account of exception cases and parameterized types at specification level. The features of genericity and exception handling have been determinant in the selection of Ada as the implementation language, but our approach may be similarly applied to other languages holding those features.

This work is structured into three main sections, besides this Introduction and the Conclusion. Section 2 contains the main features and building primitives of the Pluss Specification Language. As an example we present the array abstract data type specification, which will be "reused" in the next sections. Section 3 describes the mechanism followed in order to "translate" form Pluss to Ada; a complete example illustrating this process for the array data type is furnished. The structure of the proposed Ada Library is discussed. Finally in Section 4 the SPADA tool, for the partial automatization of this development process, is presented.

Some basic concepts on algebraic specifications of abstract data types will be introduced, in order to clarify the further reading.

Algebraic specifications of abstract data types. Basic concepts.

An algebraic specification [11], [12] defines a class of algebras, also called *models*, that is to say a set of operations on various sets of values. An *algebra* is just a possible *implementation* of the sorts and operation names which occur in the specification. A *data type* is characterized by one or more sets of values and by the operations performed on those values. The algebraic approach defines a data type as a *many-sorted algebra*, which is constituted by one or several sets of values, called *carriers*, and some operations defined on these sets. These sets are named, since it is necessary to distinguish them, and their names are called *sorts*. The *signature* Σ of a data type is constituted by its sorts, the names and *arity* of its operations, that is to say the sorts of their domains and co-domains. Some of the operation names are *constant*, with only a co-domain, meaning that the corresponding operation has no operands. Given a signature Σ, the many-sorted algebra, called Σ-*algebra*, is an algebra where sets and operations are named following the names of Σ. It corresponds to any implementation of the names of Σ. A Σ-*term* is any valid composition of sorted variables and operations of Σ. If the considered Σ-algebra is partial, the operations will be partial ones. We will be dealing with algebras where the operations are total ones, called E,R-algebras (Error, Recovery algebras), following the formalism of [4], which are Σ-algebras where the carriers are splitted into two disjoint sub-sets constitued by erroneous values and non-erroneous values. Exceptions and their eventual recovery are specified by means of two subsets of *declarations* . The E,R-signature is constituted by adding to Σ these two subsets. This formalism is

particularly useful because it considers exception handling, thus allowing a more realistic specification of data types. Other formalisms have been established since then in order to cope with exceptions [5], [6], but what is important is to be able to specify exceptions and not much what formalism has to be used. An *abstract data type*, which will be denoted by ADT, is a class of many-sorted algebras with the same signature and some specified common properties. In our framework, we will only consider finitely generated algebras w.r.t. the declared set of generators. An *algebraic abstract data type* is the definition of an ADT by means of a signature and a set of *axioms*. In the case of E,R-algebras, the axioms are separated into *ok-axioms* and *er-axioms*. In our context we will be using only ok-axioms and exception declarations.

2. THE PLUSS ALGEBRAIC SPECIFICATION LANGUAGE.

The Pluss (a Proposition of a Language Usable for Structured Specifications) language [10] provides a powerful way of *structuring* algebraic specifications. The simplified $ALEX_{PLUSS}$ version of Pluss [7] and [3], which includes exception handling facilities, will be used here.

2.1 The Pluss Building Primitives.

The SPEC (specifications where the class of possible implementations is fixed) are obtained by:
- an enrichment (USE keyword) , - an instance of a parameterized specification, - the fixed form of a DRAFT (specification component under design).
The BASIC SPECs are SPECs which don't enrich any other component. In this work we will be interested in SPECs components obtained by enrichment and instantiation.

2.2 Parameterization and Instantiation.

Parameterization allows the use of generic specifications, hence saves writing as many specifications as instances of a given specification are required. Parameterization involves three different entities: - a *parameterized specification* (denoted by a SPEC with the formal parameter list in brackets), - the *formal parameter specification* (keyword PAR) which include only one module. It specifies those minimal properties which have to be satisfied by the formal parameters and which have to be retained at the moment of instantiation, - the *instantiation mechanism* which consists in substituting a convenient specification (an actual parameter) to the formal parameter specification, in order to obtain the specialized version of the parameterized specification.
As an example, we will show below the generic specification of the array data type, parameterized by the INDEX and ELEM formal parameters:

```
SPEC : ARRAY [INDEX,ELEM]
SORTS : array
GENERATED BY :
init :              index index -> array
assign :            array index elem -> array
```

```
OPERATIONS :
lwb :                 array -> index
upb :                 array -> index
access :              array index -> elem
sub_array :           array index index -> array
EXCEPTION CASES :
illegal_access :            or(i<lwb(t),i<upb(t)) => access(t,i)
illegal_assign :            or(i<lwb(t),i<upb(t)) => assign(t,i,v)
illegal_init :        i>j => init(i,j)
illegal_sub_array :   or(or(i>j,i<lwb(t)),j>upb(t)) => sub_array(t,i,j)
OK-AXIOMS :
bound1:       lwb(init(i,j)) = i
... ... ...
access1:      i = j => access(assign(t,i,v),j) = v
... ... ...
sub1:         sub_array(init(i,j),k,l) = init(k,l)
... ... ...
WHERE :
t : array
i,j : index
v,v' : elem
END ARRAY
```

The INDEX formal parameter specification is the following:

```
PAR : INDEX
USE : INTEGER
SORTS : index
OPERATIONS :
first :         -> index
last :          -> index
_ : index       -> integer
EXCEPTION CASES :
first-1 : pred(first)
last+1 :        succ(last)
END INDEX
```

Note that the operation "_" is a coercion of index into integer. Thus any operation defined on integers can be applied as well on indexes (such as e.g. addition, etc.).

The ELEM formal parameter specification is expressed as follows:

```
PAR : ELEM
USE : BOOL
SORTS : elem
OPERATIONS :
eq : elem elem -> bool
OK-AXIOMS :
eq1 : eq(x,x) = true
eq2 : eq(x,y) = eq(y,x)
eq3 : and(eq(x,z) = true, eq(z,y) = true) => eq(x,y) = true
WHERE :
x,y : elem
END ELEM
```

Notice that PAR ELEM is a short specification, contained in only one module, and sufficiently general to be used as a formal parameter in other generic specifications. The specification corresponding to the data type may be seen in [15] .

3. THE ADA COMPONENTS.

In this section we will describe the mechanism involved in the development of an Ada component. We observe two abstraction levels: the specification level (the ADT is expressed in Pluss) and the implementation level (the ADT is expressed in Ada). The "translation" mechanism for passing from one level to the other and the structure of the Ada Library are described.

3.1 Translation of algebraic specifications.

We will now introduce the steps required in order to "translate" a Pluss specification into the Ada language constructs. In the following paragraphs the cases of an ordinary (non parameterized) and parameterized specification will be discussed.

3.1.1 Ordinary specifications.

We define a *translation* function ρ_S : S -> P, where S is a Pluss specification and P is an Ada package, which maps all the elements of the signature of S (sorts, operations and exception declarations) into the package specification part (data types, functions or procedures and exception declarations), corresponding to the P package in Ada. That is to say that an operation F of S, whose arity is F : TS -> TE, may be translated by ρ_S into an Ada function *f(... : in TS) return TE;* of package P. We can observe that $\rho(F)$ is not exactly equal to f, which is denoted by $\rho(F) \approx f$, because in order to obtain f we have to take into account the Ada language syntactical constructs (reserved words, special symbols, etc.). We will also assume that the elements of S will be translated by ρ_S into valid Ada identifiers. For example, in the specification of SPEC ARRAY, the sort *array* will be renamed into the *p_array* type and the specification name ARRAY will be renamed into the package identifier *sp-array*, because *array* is an Ada predefined type. Similarly, the operation *access* will be renamed into the *acces* function name identifier, because *access* is an Ada reserved word indicating an access type.

3.1.2 Parameterized specifications.

In this case, two aspects have to be considered: the translation of a generic specification into a generic Ada package and the instantiation of the corresponding formal parameters.

3.1.2.1 Translation.

This situation is similar to the non parameterized case, but two translation functions have to be defined:

1. ρ_{FPS} : FPS -> GPP, which is a mapping of the formal parameter specification FPS of a generic specification S_g into the generic part GPP of a generic package P_g.

2. ρ_{GS} : GS -> GP, which maps the signature of S_g into the generic package specification P_g.

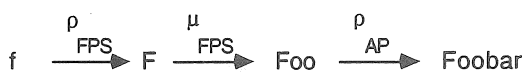
The composition of both functions ρ_{FPS} and ρ_{GS}, plus the Ada language syntactical constructs, will represent the translation of the Pluss generic specification S_g into the Ada generic package P_g, that is to say: $(\rho_{FPS}+\rho_{GS})\ (S_g) \approx P_g$.

3.1.2.2 Instantiation.

We consider, as before, the situation in which the formal parameter specifications of S_g are mapped into the generic part of the generic package P_g by function ρ_{FPS}. We have now to consider an *instantiation morphism* μ_{FPS} : FPS -> AP_{S_g} which takes the formal parameters of S_g and maps them into the corresponding actual parameters. Let us take an operation F of the FPS of S_g and the corresponding operation Foo of the actual parameters, we have $\mu_{FPS}(F)$ = Foo.

Even if there are some differences between the Pluss and Ada instantiation mechanisms, we can define a function ρ_{AP} : AP_{S_g} -> AP_{P_g}, establishing a correspondence between the actual parameters AP_{S_g} of the generic algebraic specification S_g and the actual parameters AP_{P_g} of the generic package P_g.

By means of the instantiation morphism μ_{FPS} and the functions ρ_{FPS} and ρ_{AP}, we are able to establish a correspondence between an operation of the generic part of the P_g package and an operation of its actual parameters AP_{P_g}. Taking an operation f in P_g and an operation Foobar of AP_{P_g} we will have:



3.1.2.3 An example: translation and instantiation mechanisms for the array data type.

This example illustrates in the first place the translation of the ARRAY generic specification, shown in Section 2.2, into the Ada generic package *sp_array*, by means of the translation function. In the second place, the instantiation of the formal parameters of SPEC ARRAY, by means of the instantiation morphism, will be illustrated showing also the obtention of the instantiated *sp_array* generic part.

Translation and instantiation of the parameterized specification SPEC: ARRAY.

In order to describe this process, we show below the Ada code corresponding to the specification part of the generic package *sp_array*. The algebraic specification used in the example is the SPEC ARRAY.

```
generic
  type index is (<>);
  type elem is private;
package sp_array is
  illegal_acces, illegal_assign, illegal_init, illegal_sub_array :  exception;
  type p_array is private;
function init (i,j:index) return p_array;
function lwb (t:p_array) return index;
```

```
function upb (t:p_array) return index;
procedure assign (t:in out p_array; i:index; x:elem);
function acces (t:p_array; i:index) return elem;
procedure sub_array (t:in out p_array; i,j:index);
private
  type arr_elem is array (index range <>) of elem;
  type p_array is record
                    val : arr_elem(index);
                    pri : index;
                    ult : index;
                  end record;
end sp_array;
```

We will apply function ρ_s in the first place in order to translate the algebraic specification name, the names of the exceptions and the sort into the corresponding elements of the Ada package:

ρ_s(SPEC: ARRAY) ≈ package sp_array is

ρ_s(EXCEPTION CASES :
illegal_access :
illegal_assign :
illegal_init :
illegal_sub_array :) ≈ illegal_acces, illegal_assign, illegal_init, illegal_sub_array: exception;

ρ_s(SORTS: array) ≈ type p_array is private;

Notice that, in order to clarify our discussion, we will show as domain and co-domain of the translation functions the syntactical constructions of languages Pluss and Ada.
Now, function ρ_{FPS} will be applied to the formal parameters INDEX and ELEM of SPEC ARRAY in order to obtain the generic part of the *sp_array package* .

ρ_{FPS}(INDEX, ELEM) ≈ generic type index is (<>); type elem is private;

Function ρ_{GS} is now applied to the generators and operations of the signature of SPEC: ARRAY :

ρ_{GS}(init : index index -> array) ≈ function init (i,j: index) return p_array;

ρ_{GS}(lwb : array -> index) ≈ function lwb (t: p_array) return index;

ρ_{GS}(upb : array -> index) ≈ function upb (t: p_array) return index;

ρ_{GS}(assign : array index elem -> array) ≈ procedure assign (t: in out p_array; i: index; x:elem);

ρ_{GS}(access : array index elem -> array) ≈ function acces (t: p_array; i:index) return elem;

ρ_{GS}(sub_array : array index index -> array) ≈ procedure sub_array (t: in out p_array; i,j:index);

The translation of the signature and formal parameters of the specification SPEC ARRAY is then obtained combining respectively all the applications of ρ_s , ρ_{FPS} and ρ_{GS}. We assume that $S_G = S_{G1} \cup S_{G2} \cup FPS$. For ρ_s we have : $\rho_s(S_{G1}) = \rho_s$(SPEC: ARRAY) + ρ_s(EXCEPTION CASES: illegal_access: illegal_assign: illegal_init : illegal_sub_array:) + ρ_s(SORTS: array),
and respectively for ρ_{GS}

$\rho_{GS}(S_{G2}) = \rho_{GS}$(init : index index -> array) + ρ_{GS}(lwb : array -> index) + ρ_{GS}(upb : array -> index) + ρ_{GS}(assign : array index elem -> array) + ρ_{GS}(access : array index -> elem) + ρ_{GS}(sub_array : array index index -> array)

In the case of ρ_{FPS}(FPS) we have only the application to the parameters INDEX, ELEM.

We finally obtain: $(\rho_s + \rho_{FPS} + \rho_{GS})(S_G) = P_G$

Since ρ_s, ρ_{FPS} and ρ_{GS} are bijective functions, we may apply their respective inverse functions ρ_s^{-1}, ρ_{FPS}^{-1} and ρ_{GS}^{-1} to the package P_G in order to obtain the algebraic specification S_G :

$(\rho_s^{-1} + \rho_{FPS}^{-1} + \rho_{GS}^{-1})(P_G) = S_G$

In order to describe the instantiation mechanism, we consider the same function ρ_{FPS} to map the parameters FPS into the package generic part GPP. The instantiation morphism μ_{FPS} is then applied in order to map the formal parameters FPS into the actual parameters AP:

ρ_{FPS}(INDEX, ELEM) ≈ generic type index is (<>); type elem is private;

Reciprocally we have: ρ_{FPS}^{-1}(generic type index is (<>); type elem is private) ≈ INDEX, ELEM;

We apply now the morphism μ_{FPS} to FPS in the specification S_g in order to obtain respectively sort INTER (algebraic specification for an interval) and sort INTEGER as the actual values, in the example, of parameters INDEX and ELEM : μ_{FPS}(INDEX) ≈ INTER and μ_{FPS}(ELEM) ≈ INTEGER.

Function ρ_{AP} is now used to establish the correspondence between the actual parameters of the generic algebraic specification and the actual values of the generic part of the package. Substituting the values for μ_{FPS} we obtain the required correspondence between both instantiations:

$\rho_{AP}(\mu_{FPS}$(INDEX), μ_{FPS}(ELEM)) = ρ_{AP}(INTER, INTEGER) ≈ type inter is range 1..6; package tab_int is new sp_array(index => inter, elem => integer);

Notice that the generic package *sp_array* is then instantiated by the *inter* and *integer* types, as actual values of the parameters *index* and *elem* respectively. Notice also that the name of the instantiated generic package is *tab_int*.. Since we have that $\rho_{AP}(\mu_{FPS}(FPS)) \approx AP_{P_g}$, we may automatically obtain the instantiations *inter* for *index* and *integer* for *elem* corresponding to the generic part of package P_g.

Notice that the approach followed up to now is sufficiently general, in the sense that it may applied to any implementation language, providing it holds genericity and exception handling features.

3.2 The structure of the Ada Library.

We have been inspired by the work of [9] for the structure of our Library. We will group the data types into four main classes:

1. The **building types** class. These data types allow the combination of basic types (integer, boolean, ...), in order to construct more complex data types (tree, graph, ...). We will include in this class the *array* and *record* data types.
2. The **sequential types** class. These data types are basically represented by the linear lists, containing *elements* which have to be processed sequentially. We will consider the *sequence* and *property list* data types.
3. The **tree-like types** class. These data types are constituted by sets of elements called *nodes*, hierarchically organized, with a distinguished node called *root*. The *binary tree* and *general tree* data types will represent this class.
4. The **relational types** class. These data types consist of a set of objects called nodes and relations among these objects called arcs or edges. The *graph* data type is representative of this class.

We have selected commonly used representations to implement these types. The complete Ada Library may be seen in [15]. As an example, the specification part of the array data type was shown in Section 3.1.2.3. The Ada components have been built from the corresponding Pluss specification, as it has been shown in Section 3.1. The SPADA tool is used in order to automatize the derivation of the specification part of the package and it will be described in Section 4.

4. THE DEVELOPMENT OF THE ADA COMPONENTS.

The SPADA [13] tool used for directly deriving an Ada package skeleton from its algebraic specification will be briefly described in this section. We have to mention that this tool is integrated into a prototype Ada Support Environment, system M-APEX [14], built at the ISYS research center, Caracas, which will not be detailed here. The M-APEX support environment is written in Golden Common Lisp and runs on microcomputers PC, PS or compatibles under MS-DOS. It offers several tools supporting specification and implementation as early stages of software development. Hence two different languages are supported, corresponding to the two abstraction levels of development considered: a specification language [1], [17] and a high level programming language [13]. The environment is highly interactive and the integration of the tools is seen as a way of grouping them around the abstract data type as a common or unifying concept. The tools communicate with each other consulting or updating the Knowledge Base. Internal forms are kept for the different objects manipulated within the environment.

The SPADA tool.

The SPADA tool has been inspired in the work of [18]. It generates automatically a skeleton of an Ada package directly from the corresponding Pluss specification. The skeleton is derived from the ADT signature, taking account of domains and co-domains of each operation, USEd ADT

specifications and exception declarations. SPADA produces the header and the function body skeleton for each operation. In the skeleton, the data type representation, the variables of the function header and the function bodies are missing and they have to be completed by the user. Since at Ada level the user may require some functions to be implemented as procedures for efficiency purposes, an interactive dialog is established in order to point out what are those operations that will be implemented in Ada as procedures. A small editor is also provided in order to interactively fill in the representation for the package that is being derived. The representation is actually limited to standard types and in the SPADA actual version does not allow to add type instantiations. An Ada internal form is also kept for the package. The SPADA tool also allows the integration of the function bodies which are built within the APA tool [2], in order to complete the package skeleton. In this way an executable code may be obtained (actually ADAC, the M-APEX Ada syntax driven editor [16], may be also used in order to eventually complete the type representations, instantiations and the function bodies). Nevertheless, no checkings are performed with respect to the sort and operation names. They are directly taken from the algebraic specification, in the sense that the user must take care, while specifying, not to use Ada reserved words, which may be Pluss valid identifiers.

5. CONCLUSION.

The aim of this work was to show the possibility of practical use of algebraic specifications for systematic derivation of Ada code. Moreover, the approach followed for the "translation" from the specification language to the implementation language is independent from the language used. We are not synthetizing nor prototyping programs. We are situated in the domain of assisted program construction, where the transformation of the program are mostly user-directed. Special care has been taken in the consideration of exceptions, in order to guarantee the robustness of the resulting programs. All the programs produced have been tested on an Alsys Ada compiler at the Laboratoire de Recherche en Informatique (L.R.I.), Orsay, France. Parameterized specifications make easy the reuse of the components. However, we have to point out that the algebraic specification is used only as a guide to the program construction. The axioms are not directly involved in the process. Nevertheless this fact does not minimize their fundamental importance for comprehension of the essential properties relative to the program to be constructed. The Ada Library is actually constitued by few and very well known components; we hope to extend it including more complex ones and have a choice of different representations for each component.

6. ACKNOWLEDGMENT.

The author is greatly indebted to Michel Bidoit for its always pertinent advice and fruitful remarks and to Alfredo Matteo and Françoise Schlienger, who have followed very closely this research.

7. REFERENCES.

[1] E. ACOSTA "SPEC-F: Un Formateador de Especificaciones Algebraicas expresadas en el lenguaje Pluss." Tra. esp. de Grado para la Licenciatura en Computación, UCV, Agosto 1988.

[2] L. ANES, J. HERNANDEZ "Un sistema de construcción automática de programas Ada: implementaciónde los módulos de construcción." Trab. esp. de Grado para la Licenciatura en Computación, UCV, Marzo 1991.

[3] M. BIDOIT "Pluss, un langage pour le développement de spécifications modulaires." Thèse d'Etat, Université Paris-Sud, Orsay, Mai 1989.

[4] M. BIDOIT "Algebraic specification of exception handling and error recovery by means of declarations and equations." Proceedings of the 11th. Int. Colloquium on Automata, Languages and Programming (ICALP), pp.95-108, Springer Verlag L.N.C.S. 172, 1984.

[5] G. BERNOT "Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs: application à l'implémentation et aux primitives de structuration des spécifications formelles." Thèse de 3ème cycle, Université Paris-Sud, Orsay, 1986.

[6] G. BERNOT, P. LE GALL "Label algebras and exception handling." Rapport de recherche L.R.I. Orsay, No. 712, 12/91

[7] F. CAPY "ASSPEGIQUE: un environnement d'exceptions ... ..." Thèse de 3ème cycle, Université Paris-Sud, Orsay, 1987.

[8] DEPARTMENT OF DEFENSE OF THE USA. "Reference Manual for the Ada Programming Language." ANSI/MIL-STD 15-A, January 1983.

[9] C. FROIDEVAUX, M.C. GAUDEL, M. SORIA "Types de données et algorithmes." Mac Graw Hill, Paris, 1990.

[10] M.C. GAUDEL "A first introduction to PLUSS" Rapport interne, L.R.I., Orsay 1984.

[11] J.A. GOGUEN, J. W. THATCHER, E G. WAGNER "An initial algebra approach for the specification, correctness and implementation of abstract data types." Vol. 4 of Current Trends in Programming Methodology, Prentice Hall 1978.

[12] J.V. GUTTAG "The specification and application to programming of abstract data types." Phd Thesis, University of Toronto, 1975.

[13] S. HANDAM "SPADA: Generador de un paquete Ada a partir de una especificación algebraica de un tipo de dato abstracto." Trab. esp. de Grado para la Licenciatura en Computación, UCV, Julio 1991.

[14] F. LOSAVIO "Towards the Construction of Interactive Working Environments: A prototype Ada Environment." Proceedings of the ICNTSSD'89, Caracas, November 1989.

[15] F. LOSAVIO "Dérivation systématique de programmes Ada comportant des tratements d'exception à partir de spécifications algébriques de types de données." Thèse de Doctorat, Université Paris-Sud, Orsay, Novembre 1991.

[16] Z. MILANO "ADAC: Un Editor Sintáctico Ada parta Microcomputadores tipo PC." Trabajo especial de Grado para la Licenciatura en Computación, UCV, Mayo 89.

[17] F. SANTAMARIA "Editor SPEC-E: Un Editor Sintáctico para el Lenguaje de Especificaciones Pluss-E." Trab. esp. de Grado para la Licenciatura en Computación, UCV, Junio 89.

[18] F. SCHLIENGER-DUPUY "Un environnement de programmation Ada intégrant des spécifications algébriques." Thèse de 3ème cycle, Université Paris-Sud, Orsay, Février 1984.