ADA CODE FROM ALGEBRAIC SPECIFICATIONS WITH
REWITING SYSTEMS\textsuperscript{1}.

E. ACOSTA* , F. LOSAVIO
Centro de Ingeniería de Software Y Sistemas ISYS, Escuela de Computación,
Universidad Central de Venezuela, Apdo. 47567, Caracas, Venezuela.
e-mail: eacosta@conicite.ve, flosavio@conicite.ve

Abstract
Formal specifications arise today as an important issue towards the development of reliable software. In
this work, algebraically specified abstract data types are used to generate a rapid prototype, using a
compilation process. Ada is the target language for the prototype, which is automatically obtained from
the axioms of the algebraic specification by means of rewriting systems techniques. A system
implementing this approach has been built and integrated into an existing algebraic specification
environment.

1. INTRODUCTION.
The main goal of this work is to present the generation of a rapid prototype, expressed in Ada\textsuperscript{2}
[DoD83], from algebraic specifications of abstract data types by means of a compilation process. This
approach is implemented by a system ADARES: ADA programs from REwriting Systems.

An aim of the specification stage is to establish the definition of the problem to be solved. Informal
specifications have the advantage to be easily written, nevertheless they may cause troubles throughout
the whole software life cycle, such as missing requirements, ambiguity, lack of coherence and
incompleteness of the information. This fact increases the effort required in later stages of software
development, such as implementation, testing and maintenance. Formal specifications give a precise
and non-ambiguous definition of the problem to be solved and they are useful to produce rapid
prototyping.

This paper is structured in four sections, besides this introduction and the conclusions: Section 2
describes the contexts of algebraic specifications and rewriting systems. Section 3 presents the
ADARES system. Section 4 is dedicated to the process for code generation (rapid prototyping) and
finally Section 5 shows the integration of the ADARES system into an existing specification
environment.

2. ANTECEDENTS.
The approach followed in this work relies on the formal bases of algebraic specifications and rewriting
systems, which will be briefly presented through the next sections.

2.1 The algebraic specifications' context.
Algebraic specifications [Gut75], [GTW78] allow to define an abstract data type as one or more sets of
values (called sorts) and operations on these sets. The sorts with the names of the operations and their
arities (sorts of the domain and codomain) constitute the signature of the data type (denoted by \( \Sigma \)). In
the set of operations defined on a data type, we distinguish the constructor operations, for generating
all the possible sort values. Those operations whose domain is empty are considered as constants.
Given a signature \( \Sigma \), the many-sorted algebra \( \Sigma\)-algebra, is an algebra where sets and operations
are named following the names of \( \Sigma \). It corresponds to any implementation of the names of \( \Sigma \), constituting
a model of the names of \( \Sigma \). A \( \Sigma \)-term is any valid composition of sorted variables and operations of \( \Sigma \).

Actually, several algebraic specification languages and the corresponding environments have been
defined [Gau84], [Bid89], [G&W88], [GHW85]. In this work our interest is focused on the
ASSPEGIQUE [Cap87] environment supporting the PLUSS specification language. The algebraic
specifications will be expressed in a subset of this language and its syntax is illustrated by the
following example, showing a PLUSS non parameterized or ordinary specification corresponding to a
stack of natural numbers:

\textsuperscript{1}This research is partially supported by the Consejo de Desarrollo Científico y Humanístico (CDCH) of the
U.C.V. and the French Postgraduate Cooperation Program in Informatics of the CEFI-CONICIT.
\textsuperscript{2}Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.
spec STACK;
use NATURAL;
sorts Stack;
generated by
  empty: \rightarrow Stack;
  push: Nat \times Stack \rightarrow Stack;
operations
  pop: Stack \rightarrow Stack;
  top: Stack \rightarrow Nat;
  height: Stack \rightarrow Nat;
predicates
  isempty: Stack;
preconditions
  top(x) is defined if \text{isempty}(x) = false;
axioms
  pop-1: pop(\text{empty}) = \text{empty};
  pop-2: pop(\text{push}(x, p)) = p;
  top: \text{top}(\text{push}(x, p)) = x;
  isempty-1: \text{isempty}(\text{empty}) = true;
  isempty-2: \text{isempty}(\text{push}(x, p)) = false;
  height-1: \text{height}(\text{empty}) = 0;
  height-2: \text{height}(\text{push}(x, p)) = s(\text{height}(p));
where
  x: Nat; p: Stack;
end STACK;

Example 1. Ordinary specification of the Stack abstract data type

Moreover, the PLUSS specification language allows to write \textit{parameterized} or \textit{generic} specifications. A generic specification involves three elements: the \textit{parameterized} specification, the \textit{formal parameter} specification, and the \textit{actual parameter} specification. The \textit{instantiation} process consists in substituting the formal parameter specification by the convenient actual parameter specification, in order to obtain the specialized version of the parameterized specification. Example 2 below shows a generic specification corresponding to a stack of elements.

generic spec GEN_STACK[ELEMENT];
use NATURAL;
sorts Stack;
generated by
  empty: \rightarrow Stack;
  push: elem \times Stack \rightarrow Stack;
operations
  pop: Stack \rightarrow Stack;
  top: Stack \rightarrow elem;
  height: Stack \rightarrow Nat;
predicates
  isempty: Stack;
preconditions
  top(x) is defined if \text{isempty}(x) = false;
axioms
  pop-1: pop(\text{empty}) = \text{empty};
  pop-2: pop(\text{push}(x, p)) = p;
  top: \text{top}(\text{push}(x, p)) = x;
  isempty-1: \text{isempty}(\text{empty}) = true;
  isempty-2: \text{isempty}(\text{push}(x, p)) = false;
  height-1: \text{height}(\text{empty}) = 0;
  height-2: \text{height}(\text{push}(x, p)) = s(\text{height}(p));
where
  x: elem; p: Stack;
end GEN_STACK;

par ELEMENT; sorts elem; end ELEM;

Example 2. Generic specification of the Stack data type and parameter specification

Example 3 below shows the instantiation corresponding to the generic Stack data type:
spec NAT_STACK;
as
  GEN_STACK(ELEMENT \Rightarrow NATURAL by m); where m: elem \Rightarrow Nat;
end
NAT_STACK;

Example 3. Instantiation of the generic Stack data type
2.2 The rewriting systems' context.
An important goal in the manipulation of modular and hierarchical specifications is to convert them into executable specifications (a program for example), by means of a preferably automatic translation process. The executable specifications may be used as prototypes of the system to be constructed. The prototype allows to check the behavior of the system using a symbolic evaluator. The prototype used in this way constitutes, in some cases, an excellent mean to check the consistency and completeness of a specification, verifying whether the specification accomplishes its expected behavior. This does not mean that either the prototype "replaces" the specification or vice versa. Both steps, specification and prototyping, have to be included in the software life cycle. In our context, they are considered in this order since the prototype is generated from the specification.

The rewriting technique consists basically in giving an orientation to the equational axioms. Once the orientation has been selected, the axiom is converted to an oriented equation, called rewriting rule, denoted by \( t \rightarrow t' \). A set of rewriting rules, where the set of variables of the right term \( t' \) is included in the set of variables of the left term \( t \), constitutes a Terms Rewriting System (TRS). Rewriting a term means to replace a subterm, which is made equal to the left hand side term of a rewriting rule, with the right hand side term of the rule, whose variables may be affected by a certain value (by a substitution) during this process. We say that \( t \) is rewritten or reduced into \( t' \), denoted by \( t \rightarrow_R t' \), where \( R \) is a TRS, iff:
1. There exists a subterm \( u \) of \( t \).
2. There exists a rewriting rule \( l \rightarrow r, r \in R \)
3. There exists a substitution \( \pi \) such that when it is applied to \( l \), it results a term equal to \( u \), that is to say \( \pi(l) = u \)
4. \( t' \) is obtained replacing the subterm \( u \) of \( t \) resulting from the application of the substitution \( \pi \) to \( r \), that is to say \( t' = t[ u \leftarrow \pi(r)] \).

The iteration in this process, using a rewriting system \( R \), is called reduction or rewriting. Each rewriting step establishes a relation between two terms, denoted by \( \rightarrow_R \). If two terms are rewritten into the same term, a rewriting proof is obtained. When a term cannot be rewritten, it is said to be in its normal form. A rewriting system is said to be natherian (terminating) if every term has a normal form and it is said confluent if the reduction of every term, independently of the applied rule, generates the same normal form. Hence, a rewriting system is said to be canonical or convergent if it is natherian and confluent.

It is possible to associate a rewriting system to the set of axioms of an algebraically specified abstract data type, allowing the evaluation of the expressions computing its normal form through a completion mechanism. If the initial set of axioms is formed only by universally quantified equations, the completion process determines the (arbitrary) orientation of the axioms from left to right. Now, if the axioms of a specification are restricted only to these kind of equations, the semantic expressive power of the specified data type decreases. Besides, it may not be sufficient to orient the equations from left to right, because if we want to obtain rewriting rules, all the variables in the right terms of the rules must appear in the left terms. Moreover, the obtained rewriting system is expected to be terminating, confluent, and defining the same equational theory as the original set of equations. A Knuth-Bendix [KB70] algorithm is used in order to compute the completion process, determining from a set of equations, a terminating and confluent rewriting system. Several systems such as REVE [DF84], AFFIRM [Mus79] and KB[Fag84] have been built to implement this process.

3. THE ADARES SYSTEM.

3.1 General Description
The ADARES system uses algebraic specifications in order to produce a rapid prototype of the specified system. This prototype is obtained compiling the TRS associated to the set of axioms of the specification. It is supposed that the TRS meets the confluence and termination properties. Moreover ADARES allows the introduction of valid expressions with respect to a compiled algebraic specification, in order to compute their respective normal forms by symbolic evaluation. The compilation of an algebraic specification means code generation in a target language and it is used to indicate the process of computing the normal form.

Ada is the target language of the ADARES compiler. The Ada language has been selected because it provides the features of strong typing, modularity, encapsulation and genericity, facilitating the translation of an algebraic specification expressed in a specification language, into a high level implementation language.
The prototype is built by ADAES in two steps: 1) A convergent rewriting system, corresponding to an algebraic specification expressed in PLUSS, is analyzed. 2) A compilation process is applied to the rewriting system. An Ada package, which will be used to evaluate the normal forms, is generated. The complete compilation process will be described in Section 4.

3.2 The specification of the ADAES compiler.
The algebraic specification approach has also been used here to specify the main part of the ADAES system, its compiler. The algebraic specification describes precisely each one of the modules constituting the compiler. These modules are: the Ada package used to evaluate the normal forms (ADA-EF); the specification library (LIB); the algebraic specification internal representation of ASSPESQUE (ASS-REP). In what follows we will show as an example, the algebraic specification diagram corresponding to ADA-EF which is the Ada package generated during the compilation process for evaluating the normal forms (see Figure 1). It contains the import constructs making visible other Ada units, defined into the WITHUSE module, which also requires the FISTLIB module. The package specification part is represented by module PACKSPEC, constituted by modules DCLTYPE (specified sorts or declarations of private types), DECLTDES (declaration of the private constants, corresponding to constructor operations) and HEADERS (operations' headers) respectively. This module requires also of module ONEHEADER (profile of an operation). This module is also constituted by the specification module NAME (function name), PARLISE (parameter list) and module SORT (return type). All these modules require module FISTLIB. Finally the package body is represented by module PACKBODY, which is constituted by modules BODIES (procedures and functions bodies). Each body is represented by module ONEBODY, which also requires FISTLIB.

4. PROCESS FOR CODE GENERATION.
The compilation process followed by ADAES generates an Ada program which will be used for computing the normal forms. This program is constituted by a package importing other Ada units. The specification part of the package contains the following constructions: a constant for each constructor operation of algebraic specification whose domain is empty, a function for each operation defined in the algebraic specification, two functions for the treatment of coercions, a procedure for checking the input terms. The body part constitutes the implementation of the functions and procedure.

Among the program units imported by the generated package we point out the TERM package, used to implement the algebraic abstract data type, by means of a tree structure. The other units correspond to the compilation of the algebraic specification modules required by the package corresponding to the algebraic specification which is being compiled.

The rewriting rules to be translated during the compilation process are of the form \((\forall i:1..n \; ui = vi) \land (\forall j:1..m \; uj \neq vj) \Rightarrow \ t1 \rightarrow \ t2\), where all variables of \(t2\), \(ui\), \(vi\), \(uj\), \(vj\), for all \(i, j\), are in \(t1\).

The requirements for symbolic evaluation are of the form \(X=ft\) where \(t\) is an array of terms with variables. Prototyping consists in the evaluation of expressions, considering the fact that the compilation process assigns an Ada function to each operation symbol of the algebraic specification.
To solve a requirement \( X = f(t) \) is reduced to compute in Ada the expression \( f(t) \), using its standard evaluation strategy, in order to obtain the corresponding normal form. The process followed to generate the code is inspired in the work of [Kap87]:

1. \( \Sigma \) is a finite set of operation symbols containing the names of the operations defined in an algebraic specification module.
2. \( \Sigma^n \) is the set of symbols of \( \Sigma \) with arity \( n \).
3. the constants (i.e. all the symbols of \( \Sigma^0 \)) are considered to be in normal form.
4. to each symbol of the operation \( Op \in \Sigma \), there is associated an Ada function \( Op \); that is to say, if \( Op \in \Sigma^n \) then \( Op(t_1, \ldots, t_n) \in T_S \Sigma \) this expression corresponds to a term whose root is the \( Op \) symbol and its sons are \( t_1, \ldots, t_n \); whereas \( Op(t_1, \ldots, t_n) \) corresponds to the result of applying the \( Op \) function to the arguments \( t_1, \ldots, t_n \).
5. An eval function, is defined: \( T_S \Sigma \rightarrow T_S \Sigma \), associating the corresponding Ada function denoted by \( Op: (T_S \Sigma)^n \rightarrow T_S \Sigma \), with each operation symbol \( Op \in \Sigma^n \):

   \[
   \text{eval}(t) = \begin{cases} 
   \text{if } & t = Op(t_1, \ldots, t_n) \text{ then } Op(\text{eval}(t_1), \ldots, \text{eval}(t_n)) \text{ else } \{ t \text{ must be a variable } \} 
   \end{cases}
   \]

Note that when a variable appears in a term, it is considered to be in normal form. The use of variables is allowed, but they are not evaluated. They facilitate the simultaneous observation of different behaviors of the operations, since a variable represents any compatible sort term.

During the code generation process there was no need to implement the eval function. We profited from the standard Ada evaluation strategy consisting in firstly evaluate the actual parameters and secondly the function body.

The code corresponding to each operation \( Op \) is translated into Ada code, so the prototype obtained may be compiled in its turn, using any Ada compiler. This feature increases the efficiency of the prototype. ADARES is implemented in Franz Lisp [FS82]. Its platform is constituted by the ASSPEGIQUE environment whose user-interface makes use of the WI editor [AF86], based on the X protocol, under Unix\(^3\).

The process of Ada code generation is the kernel of ADARES. This process gives origin to two Ada programs for each specification module to be compiled. The first program represents the specification module to be compiled (Ada Package Program) and the second one (invocation program) corresponds to the communication between the ADARES implementation language Lisp and the target language Ada.

During the generation of the Program Ada Package, each specification module is represented by an ordinary Ada package, consisting of two parts: the package specification, providing the interface with the exterior world (with other program units, for example), and the package body, containing the implementation details. The Invocation Program is related with the Ada environment and contains the commands required to link and execute the Ada programs required for the symbolic evaluation of the expressions.

4.1 Treatment of ordinary specifications.

The strategy for the generation of Ada code involves five sequential steps, which will be described in what follows. Figure 2 below illustrates this process.

\(^3\) Unix is a trade mark of the Bell Laboratories.
i. Initial pattern for Ada code generation.
The goal of this step is the generation of the initial patterns for the Ada package and for the invocation program. These patterns contain several instructions that will appear in every generated Ada units. They will be completed through the whole process of code generation.

ii. Invocation of the required packages generation.
During this step, the sentences for importing the other packages, required by the package that is being generated, are created. These sentences correspond to the Ada with/use clauses. They are located in a separate unit. Example 4 below shows the instructions generated for the Stack specification (see Example 1).

```ada
with SEQUENTIAL_IO;
with TERMPACK; use TERMPACK;
package Term_io is new SEQUENTIAL_IO(Ident);
with Term_io;
with Text_io; use Text_io;
with TERMPACK; use TERMPACK;
with NATPK; use NATPK;
with BOOLPK; use BOOLPK;
```

Example 4. Invocation of the required packages for the Stack specification.

iii. Generation of the package specification part.
The signature of the algebraic specification is used to generate the package specification part. The following steps are performed:
a. Identification of the package. It corresponds to the name in the spec clause.
b. Declaration of the private types. They correspond to the sorts in the sort clause.
c. Declaration of the private constants. They correspond to the constructor operations with empty
domain.
d. Declaration of the functions. They correspond to the profile of the operations with non
empty domain.
e. Declaration of headers of the coercion functions: they allow to manipulate the objects of an abstract
data type specified as Term objects. Reciprocally, the conversion of a Term
object to its appropriate type (corresponding to the specified type) is also considered.
f. Declaration of the procedure for reading the expression to be symbolically evaluated.
g. Completion of the private part of the package specification: complete definitions of types and private
constants are incorporated in this step.

iv. Generation of the package body.
The generation of the package body consists in giving the complete definition of each one of the Ada
units whose headings have been generated in step iii.
Among the program units constituting the package, we distinguish two kinds:
a. Those functions corresponding to the operations defined in the specification module to be compiled.
These bodies are generated translating the corresponding axioms. In case of an operation with no
axioms, its body is constituted by the identity function.
b. The program units automatically created by the system, whose body are based in Ada predefined
functions.

v. Generation of the invocation program.
The last step in the process of code generation completes the initial pattern of the invocation program.
This program is used to symbolic evaluation of a term in order to: compile the package containing the
terms' standard definition, compile the required packages, compile the generated package, compile a
program for reading the expressions to be evaluated, link and execute the program unit corresponding to
the former program.

4.2 Treatment of generic specifications.
An ordinary (non generic) Ada package is necessary in order to generate Ada code from a generic
specification module to symbolically evaluate the expressions constructed from the operations of the
algebraic specification, obtaining the normal forms. This package may be used to check the behavior of
the generic specifications using symbolic evaluation. Two options are established in this case:
a. The code to be generated corresponds to a generic specification, regardless of its actual parameters.
This is equivalent to treat a generic specification without previous instantiation.
b. The code to be generated corresponds to a generic specification which has been previously
instantiated.
The treatment of generic specifications in both cases depends on the parameter manipulation. We
present here a possible solution to this problem.

The main functionalities of ADARES are the Ada compilation of algebraic specifications and the
symbolic evaluation of valid expressions (with variables or not) for a compiled algebraic specification.
The treatment of generic algebraic specifications without previous instantiation, consists in generating
Ada code from the generic spec and par specification modules. This code consists of an ordinary
package for each module, in order to preserve PLUSS modularity. Each one of these modules will be
considered as an ordinary specification module, except for an import clause, constituted by the with
and use Ada constructions, which will be included in the package corresponding to the generic spec
module to make visible the par module package. After generating the previous code, it is possible to
symbolically evaluate expressions using the package corresponding to the compilation of the generic
spec module.

For the treatment of generic specifications after instantiation, it is necessary to consider the modules
generic spec, par and spec. In particular, the spec module constitutes the actual parameter of the
generic specification. We proceed as before, with the difference that the correspondence between the
formal end actual parameter specification modules have to be considered while generating the package
representing the generic specification parameter.

5. ADARES WITHIN A SPECIFICATION ENVIRONMENT.
The integration of ADARES into ASSPEGIQUE increment the set of symbolic evaluation tools of
this specification environment. ADARES allows to compile specification modules into Ada and then
to evaluate expressions considering a compiled specification module.
Communication between ADARES and external entities such as ASSPEGIQUE, the Ada compiler and the user, is achieved defining levels of interface: A first interface is due to the fact that Lisp is the ADARES implementation language and this system generates and executes Ada programs. The establishment of a communication between Lisp and Ada, the Lisp-Ada interface, is then necessary. The other interface, the user-ADARES-ASSPEGIQUE interface interacts with:
- the user and the ADARES system; it allows to select a specification module and to introduce expressions. It is also used to display ADARES results.
- the ADARES and ASSPEGIQUE systems. It allows ADARES to obtain the specification modules and to save the Ada code generated into the hierarchical specification library of ASSPEGIQUE.

The communication between Lisp and Ada, shown in Figure 3 below, is needed during symbolic evaluation of expressions. When an expression is entered, it will be evaluated by the execution of the corresponding Ada function.

![Figure 3. The Lisp-Ada interface](image)

The interaction user-ADARES-ASSPEGIQUE occurs, for example, when the user selects an ADARES option; it then selects a specification module and ADARES obtains the corresponding specifications from the ASSPEGIQUE library. The user enters expressions and ADARES displays the results obtained from the Ada code execution.

There are three main modules of ADARES in charge of this interaction: the user-interface, the Ada-compilation and the Ada-evaluation modules. Figure 4 below shows the relations with these three modules and the external entities.

![Figure 4. The ADARES-ASSPEGIQUE interface](image)

6. CONCLUSION.
The main goal of this work is to design and implement a system using algebraic specifications in order to produce a rapid prototype of the specified system. Moreover, this prototype is used in order to check the behavior of this specification by symbolic evaluation of expressions. The ADARES system provides a tool for compiling into Ada a rewriting system and evaluating expressions in order to obtain their corresponding normal forms. This system has been algebraically specified using a subset of the PLUSS language and has been implemented in Franz Lisp. The most important features of the ADARES design and implementation are the terms manipulation and the Ada code generation strategy.
The treatment of the terms allows the handling of the expressions entered by the user, in order to be symbolically evaluated. The code generation strategy allows to obtain automatically a rapid prototype from an algebraic specification, expressed in a high level language. Actually, the ADARES implementation deals with specification modules constructed using an earlier version of the ASSPEGIQUE environment, providing only the enrichment primitive. It will be interesting to include into ADARES the treatment of the parameterized specifications provided in the last ASSPEGIQUE version.

7. ACKNOWLEDGMENT.
We are greatly indebted to Christine Chopy, who gave the main underlying ideas of this work. We are also grateful to Frederick Voisin and Nancy Zambrano.

8. REFERENCES.
Eleonora ACOSTA: Master in Computer Science (1993), Assistant Professor at the Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela. Research topics in: formal specifications, prototyping and automatic generation code.
E-mail: eacosta@conicit.ve

Francis LOSAVIO: received the Doctor degree in informatics in 1991 and a 3ème. Cycle Doctor degree in informatics in 1985 from the Univ. Paris-Sud, France. She also received a MSC degree in Computer Science from the Univ. Simón Bolívar, Venezuela in 1983. At present, she is Titular Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and coordinates the ISYS Research Center. Her research includes programming environments, software engineering environments, methodologies for software development, automatic program construction and formal specifications.
E-mail: flosavio@conicit.ve.