On Teaching a Data Structures and Algorithms Course through a Rigorous Approach

Favre, Liliana[1] Felice, Laura Martinez, Liliana Pereira, Claudia

Departamento de Computación y Sistemas
Universidad Nacional del Centro
de la Provincia de Buenos Aires.
7000 Tandil
Buenos Aires. Argentina.

Abstract

In this paper we describe a methodology for constructing efficient algorithms applied in an elementary course on Data Structures and Algorithms. This methodology attempts to show the essential steps in a sequential process in software development from an informally stated problem, via a formal problem specification, to a final efficient program. Students of the course are expected to have at least a year's experience in programming high level languages and elementary logic and calculus.
We describe a prototype, *AyDA,* which assists in the construction of algorithms starting from the proposed methodology.

Keywords: Data Structures and Algorithms, Algorithm Design Techniques, Formal Specifications, Programming Teaching.

1. INTRODUCTION

Often several different algorithms are available to solve the same problem. The choice of an algorithm for a particular problem can be a difficult process. Algorithm design requires to create and combine specifications and planning their implementations. There are several advantages of combining specifications:

- It describes the function of a software piece free from most implementation details.
- Because specifications can have several implementations with different performance properties, they can be used in various problems with different performance requirements.
- It allows one to make a predictive analysis of temporal and spatial complexity.

In this work a methodology applied in an elementary course on Data Structures and Algorithms is described. It integrates algebraic and imperative specifications and object-oriented languages. The outstanding features of this methodology are:

- An evolutionary development from specifications to implementations.
- The symbolic execution of specifications.
- The construction of efficient implementations reusing previous existing ones.

The methodology is based on a process that includes the following essential activities:

- Identification of the problem.
- Formalization of the problem.
- Analysis of the formal problem description.
- Construction of efficient implementations in an object-oriented language.

The formalization language is multiparadigm one, blending equational and imperative styles into an unique notation. C++ (Ellis 1990) has been chosen as the object-oriented language.

[1] CIC (Comisión de Investigaciones Científicas de la Pcia. de Buenos Aires
{lfavre, lfelice, lmartine, cpereira}@exa.unicen.edu.ar

The object classes that are involved in the problem are identified and specified in an algebraic style. This specification describes object classes in an abstract way, free from most implementation details. The object class specifications are constructed from previous existing ones by applying mechanisms provided by the algebraic language: generalization, specialization, parameterization, and instantiation. Two important relationships between classes can be distinguished: 'is-kind-of' and 'uses-a'. These relationships are connected with inheritance and client relationships in the object-oriented level respectively. The algebraic specifications are integrated with procedural schemes specified in an imperative style. They are related to different algorithm design techniques such as Divide and Conquer, Greedy, Backtracking, etc. Thus, a first version of the algorithm that combines procedural and algebraic parts is built. This version can be symbolically executed allowing us to make early validations.

Starting from object class specifications and procedural schemes it is possible to make a predictive analysis of the temporal and spatial complexity and select an implementation for each object class. Then, specifications and procedural schemes must be transformed in object-oriented code (in particular, we have experimented with C++). We can distinguish two types of transformations:

a) transformations of algebraic specifications to concrete classes in an object-oriented language.
b) transformations of procedural schemes to efficient code.

With respect to a) there exists a library of behavior specifications related to those data types most commonly used in the construction of algorithms: lists, trees, graphs, etc. The transformation is based on the application of reuse operators for renaming, restriction, composition and extension.

With respect to b) the procedural schemes can be automatically translated into C++ code.

To apply this methodology a prototype, *AyDA,* was implemented. *AyDA* provides an interactive environment, helping the students in the tasks of editing specifications, making symbolic execution, performing transformation and analyzing temporal and spatial complexity.

The paper is organized as follow. In Section 2, we give the motivation and related work. Section 3 describes the specification language LEAD. Section 4 describes the methodology and Section 5 gives the features of the implemented prototype. Finally, conclusions are made.

2. MOTIVATION AND RELATED WORK

Algorithms are often described in textbooks (Aho 1995; Baase 1988; Cormen 1990) in terms of pseudo-code or particular programming languages (C, Pascal, etc). Pseudo-codes are clearly not executable and need to be re-coded by programmers. On the other hand, algorithms in directly compilable code are closely tied to the physical structure of data they manipulate and this would not be likely to yield flexible solutions, moreover highly optimized algorithms are, in general, hard to understand.

The construction of efficient algorithms requires to start from descriptions satisfying the following conditions (Meyer 1997):

- They should be precise and unambiguous.
- They should be complete.
- They should not overspecify.

The theory of abstract data types reconciles the need for precision and completeness with the desire to avoid overspecification. They provide high-level descriptions, free of implementations concerns.

The paper (Franch 1993) describes a Programming Environment "used in teaching" designed to support symbolic execution of programs written in Merlin algebraic language. Also, many tools (Brown 1991; Gloor 1992; Ho 1993) have been developed for creating animation of algorithms that can be used to improve the learning of the algorithms. The emphasis in these approaches is on the algorithms themselves.

In our approach , the study of algorithms is not an end in itself. We intend to teach these topics in a framework that emphasizes in factors of software quality such as correctness, extensibility, reusability, efficiency, maintainability, etc.

Furthermore, as formal techniques become more and more used in the computing industry, it is important that the computing science curricula keep up with the technology trend.

3. THE SPECIFICATION LEVEL

Algebraic specifications of objects classes
Object classes can be abstractly specified by means of algebraic specifications of data types. Our approach is based on this formalism. We have define:

- A specification language LEAD, based on a subset of CIP-L language (Partsch 1990) and extended with mechanisms for constructing incomplete algebraic specifications. This mechanism allows us to specify abstract classes that will be associated with abstract classes in the object-oriented level.
- A model of reusable components that integrates algebraic specifications and concrete classes in an object-oriented language. A reusable component is a tree

that links algebraic specifications and concrete classes in an object-oriented language.

- The root is the most abstract specification and the leaves correspond to concrete classes.

Following, we describe the most relevant theoretical concepts for this work. The basic idea of the algebraic approach consists of describing data structures by just giving the names of the different set of data, the names of the basic functions and their properties which are described by equations in first-order logic. Following, we describe the syntax of LEAD specifications.

type T
export ....,s_i,..,c_j,...,f_k,...(1)
include Q1, ...
based on P1,...(2)
deferred
  sort s_i,...(3)
  s_j c_j,.... (4)
  ...
  function (s_1^k,.....,s_n^k) $s^k f_k$,.....(5)
  laws L_m
  ... (6)
effective
  sort s_i,...(3)
  s_j c_j,.... (4)
  ...
  function (s_1^k,.....,s_n^k) $s^k f_k$,.....(5)
  ... (6)
end of type.

The sequence of identifiers that follows the keyword **export** refers to the visible sorts, constants and operations provided by the type to its environment, i.e. that they can be used in other types (1).

A type is a hierarchy, i.e. it is based on other types P1 (2). This dependence is expressed by the keyword **based-on**. All the sorts, constants and visible operations of P1 can be used in the specification of T. To protect P1 however, its constituents are not visible for the types based on T unless they are listed as visible constituents of T. The primitive relationship is:

- transitive: if a type T is based on a type T' and T'' is primitive for T', then T'' is primitive for T.
- irreflexive: no type is primitive of itself.

Specifications distinguish two kinds of sections identified by the keywords deferred and effective. The **deferred** section declares operations and sorts that are not completely defined, i.e. there are not enough equations to specify the new operations or there are not enough operations to generate all values of a given sort. The **effective** section describes operations completely defined.

The signature of a type T is a triple <S,C,F> of identifiers, where:

- S: set of symbols of sorts;
- C: set of symbols of sort constants;
- F: set of symbols of operations, each operation symbol f belonging to F is associated to a functionality: f: s_1 x s_2 x.....x $s_n \rightarrow$ s. This functionality is expressed in LEAD as follows:

(s_1,s_2,.....,s_n)s f (5).

The operations can be restricted by preconditions. The visible constituents of P1 primitive types of T are part of the signature of T, too. The type **axioms** (6) are well-formed formulae of the calculus of first order predicates that define the type semantics.

The type specification can be based on sorts, constants and operations that serve as parameters. The type schemes (generic types) are denoted as the following form:

type T=(<<type parameters >>)
export (<<constituents>>)
<< body >>
end of type.

The constituents and the type body are described as it is done for non-parameterized types. The type parameters are a collection of sorts, constants and operations.

The language provides facilities to express relationships 'is-kind-of' by using the **include** clause. Many of these clauses can occur in the body of an instantiation.

The parameterized specifications can be viewed as a notational abbreviation from which specifications are generated by supplying a concrete type for the type parameters. From a parameterized specification special types can be obtained, by means of the instantiation mechanism. An instantiation can be denoted by:

type T=
  ...
include S (<<argument type>>) as (<<constituents>>)
  ...
end of type

We make the textual substitution of type parameters in the body of S by <<arguments type>> and the visible constituents of S are renamed by <<constituents>>. All primitive types of S are primitive types in T. All hidden constituents in S are hidden in T. The new type must be correct in the instantiation context. Many instantiations can occur in the body of the type specification. The occurrence order is arbitrary.

Instantiation allows specialization replacing the visible constituents of S that are not necessary by a point. Also,

it allows extensions by enriching a type with additional operations, sorts or equations.
The instantiation concept can be joined to the primitive type one; the resultant type of an instantiation can be used like a primitive type. The instantiation is denoted by:

based on (<<constituents>>) =
S(<<type-arguments>>)

Following, we partially depicts specifications of Container, List and Stack types.

```
type Container (sort data_type)
export container, init, add, is_empty, size
based on Nat, Bool
deferred
  sort container
  container init
  function(container,data_type) container add,
  function (container) nat size
effective
  function (container) bool is_empty
  laws data_type d, container c:
  (1)is_empty(init)=true,
  (2)is_empty(add(c,d))=false
end of type.
```

```
type List (sort data_type)
export  list, init, add, is_empty, length, first, rest, insert, delete, get
include Container (data_type) as ( list, init, add, is_empty, length)
deferred
  function(list l: not is_empty(l)) first,
  function(list l: not is_empty(l)) list rest
effective
  function(list) nat length,
  function(list l, data_type, nat p: 1≤p≤ length(1)) list insert,
  function(list l, nat p: 1≤p≤ length(1)) list delete,
  function(list l, nat p: 1≤p≤ length(1)) data_type get
  laws data_type d,d1, list l, nat p:
  (1)length(init)=0,
  (2)length(add(l,d))=1 + length(l),
  (3)insert(add(l,d),d1,p)= if(p = length(add(l,d)))
                    then add(add(l,d1),d)
                    else add(insert(l,d1,p),d)
                    endif,
  (4)delete(add(l,d),p)= if (p=length(add(l,d)))
                    then l
                    else add(delete(l,p),d) endif,
  (5)get(add(l,d), p)= if (p=length(add(l,d)))
                    then d
                    else get(l,p) endif
end of type.
```

```
type Stack(sort data_type)
export  stack, init, push, is_empty, length, top, pop
include Lista (data_type) as (stack, init, push, is_empty, lenght , top, pop, . , . , .)
effective
  sort stack
  stack init
  function(stack, data_type) stack push,
  function(stack s: not is_empty(s)) data_type top,
  function(stack s: not is_empty(s)) stack pop
  laws stack s, data_type d:
  (1)top(push(s,d)) = d,
  (2)pop(push(s,d)) = s
end of type.
```

Procedural schemes
The procedural schemes reflect the imperative style in a C++-like syntax. They are implemented by using a function-like encapsulation:

FUNCTION *name* (<*Argument-list*>) *result type*
    <<Body>>
RETURN *result*
END

Procedural schemes facilitate to express clearer abstractions related to algorithm design techniques (Divide and Conquer, Dynamic Programming, Greedy, Backtracking).

They support mechanisms for expressing modular design, recursion and polymorphism.

The LEAD language provides constructions for explicit binding between algebraic specifications and procedural schemes. We give an example of a function that computes the reverse of a list L.

```
function reverse_list (List L):list
{stack s;
 s=init;
 while (length(L) >0)
    {s=push (s,get(L,1));
     L=delete (L,1);
    }
 while (!is_empty(s))
   {L=insert(L, top(s));
    s= pop(s);
   }
 return L;
}
```

4. THE METHODOLOGY

The proposed methodology comprises the following stages as detailed below:

- The construction of algebraic specifications and procedural schemes.
- The symbolic execution of abstract specifications.
- The analysis of the temporal and spatial complexity.
- The transformation of algebraic specifications into concrete classes in C++.
- The transformation of procedural schemes into an imperative code C++.

Construction of algebraic specifications and procedural schemes
At this stage, a model of the final program is built to gain insight into its behavior and intended use. Object classes that are involved in the problem are specified.

For each object class a reusable component is identified. The identification of a component is correct is renaming, restriction, extension and compose reuse operators can modified it to match the object class behavior. Reuse operators on reusable components are informally defined as follows:

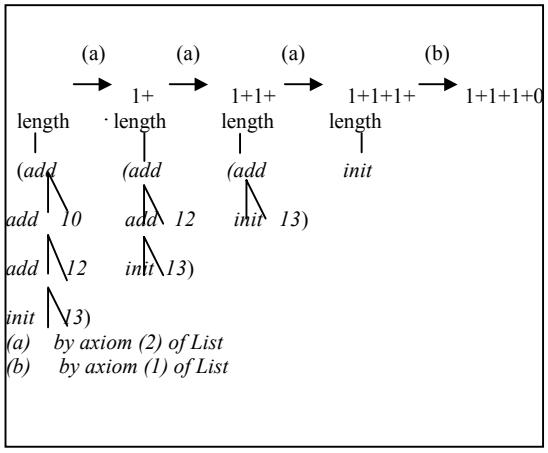
- Extend: adds sorts, operations or axioms to a specification.
- Rename: changes the name of sorts or operations.
- Restrict: forgets those parts of a specification that are not necessary for the actual application.
- Compose: combines two or more specifications in only one.

Specifications are organized in libraries to enable their selection and adaptation. This formalization increases the understanding of a system by revealing ambiguities, inconsistencies that might otherwise go undetected.

Symbolic execution
Algebraic specifications are integrated with procedural schemes, thus achieving a first version that allows us early validations by a mixed-execution mechanism.

At this stage, executions should be oriented to find out if the specification holds some properties that we know the formalization should exhibit. This allows us to correct mistakes and get a new version that will be validated again. This refinement process keeps on until the final program satisfies the initial problem requirements.

For example, the function *reverse_list* can be symbolically executed to validate behavior. Let *L=add(add(add(init,13),12) ,10)* be, the symbolic execution for: *while (length_list(l) >0)* will be:



Complexity evaluation
When a specification is going to be implemented, the first step is determining a proper representation given the operations and their efficiency constraints. The efficiency of an algorithm will depend on the choice of proper data structures. That is why this methodology, which starts from the specifications, permits to make a predictive analysis of the temporal and spatial complexity. It supposes different representations for the object classes.

The analysis of the cost has two aims: to compare the space and time requirements of an algorithm with different implementations, and to compare the requirements of two or more algorithms that perform the same function, in order to determine which is more convenient. This comparison will be made on the basis of the system resources availability (space and time) and on the importance of each resource has on the problem requirements.

The user can select the available implementations obtaining the predictive cost in space and time for the algorithm according to the selected implementation.

The C++ translation
The last step in the transformation process is the translation of formal specifications (algebraic specifications + procedural schemes) to code. Having defined formal specifications, we need to generate the C++ classes that implement the desired behavior. To achieve this we analyze every clause and relationship present in LEAD specifications and translate it into C++ code. The translation of procedural scheme to C++ executable code can be made in a semi-automatic way.

LEAD and C++ support explicit parameterization. Then, this transformation is reduced to a trivial translation. The **export** clause expresses which methods must be public, i.e. visible to other classes.

The relationship introduced in LEAD by using the clause **based-on** will be translated into a client relationship in C++. The relationship expressed through the keyword **includes** in LEAD will become an inheritance relationship in C++. In both cases we have to identify an implementation from the library of reusable components. The selected class will be transformed to obtain a new one that has exactly the specified characteristics. Possible transformations are renaming, restriction, extension and composition. It is worth pointing that this is done above the class "text", not through mechanisms provided by the C++ language. The **includes** clause may cause the application of the composition operator. When we give a list of superclasses on the algebraic level, we are just expressing that our class will have the same behavior of several other classes. This can be solved as multiple inheritance or as a composition in the implementation level.

The construction of new classes by transformation of existing ones implies access redefinition for client classes and inherited operations and, by this, the creation of an interface for the new client or superclass. Every specification can contain functions and axioms that incorporate new behavior. The application of the extension operator involves the intervention of the programmer, who should analyze different representations and provide the implementation of the new operations. The **deferred** operations will be translated into virtual methods and the **effective** ones will be mapped to concrete methods in the object-oriented level.

5. *AyDA*: THE PROTOTYPE

To apply the methodology a prototype, *AyDA*, was implemented (Martinez 1998). The prototype was implemented in Mathematica™ (Wolfram 1991). Its environment combines:

- The underlying methodology
- The algebraic specification language and procedural schemes.
- Reusable components library.

The *AyDA* architecture is summarized in Figure 1.

The main components are:

Editor: It allows specifications edition
Analyzer: It allows the syntactic analysis of LEAD specifications.
Executor System: It combines symbolic execution of procedural schemes and specifications.
Cost Evaluator: It has two sub-modules: *Temporal Cost Evaluator* and *Spatial Cost Evaluator*:

Translator: It has two sub-modules*: Procedural Schemes Translator* and *Abstract Data Types Translator.*

The system can register the 'design history' in order to give a good documentation so as to modify intermediate design decisions and maintenance. Developments are recorded automatically and they can be replay in order to accommodate changes in a reliable way.

6. CONCLUSIONS

In this paper we describe a methodology defined to support programming with abstract data types. It integrates algebraic specifications, procedural schemes and code in a rigorous framework. Specifications can be incrementally transformed into a C++ program and symbolically executed in all intermediate stages of its transformation.

We have developed a prototype, *AyDA*, for experimenting with this methodology. *AyDA* allows editing specifications, making symbolic execution, performing transformation and analyzing temporal and spatial complexity.

The methodology is applied in an elementary course on Data Structures and Algorithms and attempts to achieve the following broad aims:

- Understanding techniques to design and analyze data structures.
- Understanding algorithm design techniques (divide and conquer, greedy, dynamic programming, backtracking).
- Understanding the basis of formal specification notions.
- Learning and designing efficient algorithms for problems that use basic abstract data types (list, stack, tree, graph, etc).

Our goal is to communicate in an effective manner recurring concepts that are fundamental in Computing: levels of abstraction, reuse, complexity of large problems, conceptual and formal methods, consistency and completeness, efficiency, evolution, etc.

Our experience has shown that when students learn software quality factors early in their education, they are able to apply them along the curriculum.

Programming projects related to the course topics were developed. Empirical evaluations of our approach to teach algorithms showed that students learned algorithms better if they had hands-on experience. It is worth considering that students had submitted their projects in special tracks of conferences and some of them had been awarded.

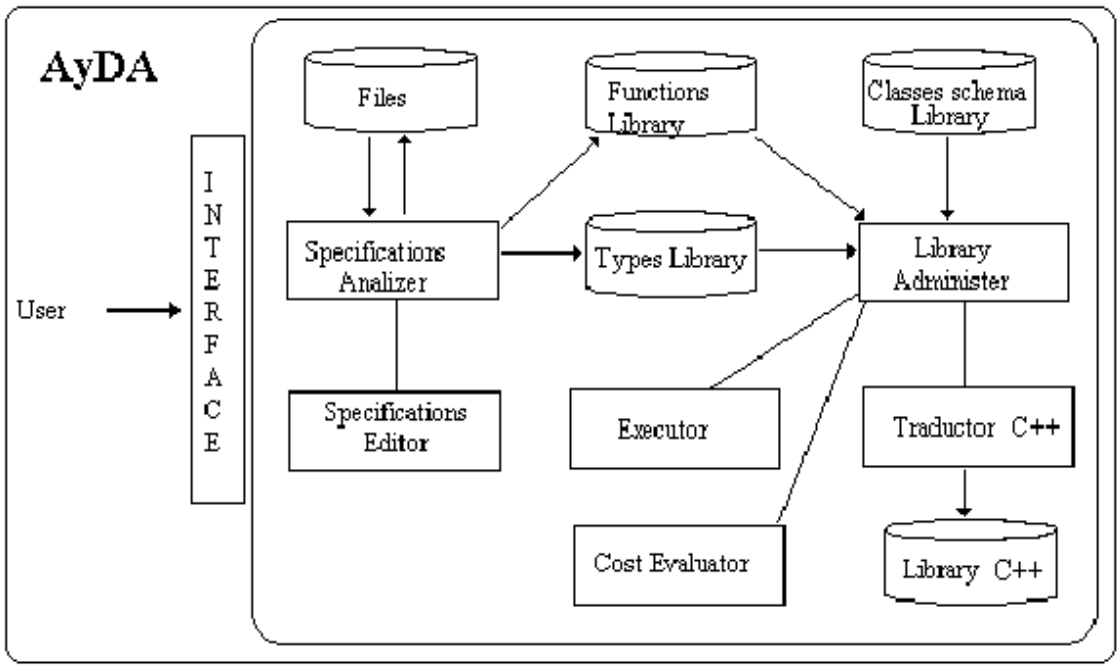


Figure 1. ***AyDA*** Architecture

Besides, we have observed that the number of promoted students in this course is high and this figure is similar to that in other second-year courses. We think that the effort to teach and introduce formal methods and object specifications is valuable, since in our approach, formal specifications are not an end in itself, they allow one to introduce abstraction mechanisms that inspire and permeate all of the curriculum and, in particular, other more practical courses such as Software Design Methodology and Object Oriented Programming.

The experience has been satisfactory. We consider that it could be reproduced in any CS/IS study program. The necessary prerequisites can be provided by introductory courses on programming, computer science, and mathematics.

7. REFERENCES

Aho, A. and J. Ullman, 1995, Foundations of Computer Science, C Edition, Computer Science Press.

Baase, S. 1988, Computer Algorithms: Introduction to Design and Analysis, 2nd Edition, Addison-Wesley.

Brown, M. 1991, "ZEUS: A System for Algorithm Animation and Multi-view Editing", Proceedings of the IEEE, Workshop on Visual Languages, pp. 4-9, Kobe, Japan.

Cormen, T.; C. Lierserson, and R. Rivest, 1990, Introduction to Algorithms, MIT Press, Cambridge, MA.

Ellis, M. and B. Stroustrup, 1990, The Annotated C++. Reference Manual, AT&T Bell Laboratories, Murray Hill, New Jersey.

Franch, X. and X. Burgues, 1993, "A case study on prototyping with specifications". ERCIM Workshop on "Development and Transformation of Programmig", Nancy, France.

Gloor, P. 1992, "AACE-Algorithm Animation for Computer Science Education", IEEE Workshop on Visual Languages, pp. 25-31.

Ho, F., C. Morgan and I. Simon, 1993, "An Advanced classroom computing environment and its applications", 24 th SIGCSE Technical Symposium on Computer Science Education, pp. 228-231.

Martinez, L. and C. Pereira, 1998, Análisis y Diseño de Algoritmos: Un enfoque a partir del paradigma transformacional, UNCPBA. Tandil. Argentina.

Meyer, B. 1997, Object Oriented Software Construction, Prentice Hall.

Partsch, H. 1990, Specification and Transformation of Programs. A Formal Approach to Software Development, Springer-Verlag.

Wolfram, 1991, Mathematica. A System for Doing Mathematics by Computer, Addison-Wesley.