STUDIA UNIV. "BABEŞ-BOLYAI", INFORMATICA, Volume I, Number 1, March 1996

A COMPILER FOR AN ALGEBRAIC SPECIFICATION LANGUAGE

D. BOZGA, D. CHIOREAN, AND I. OBER

Abstract. This paper presents an algebraic specification language called μFOOPS, the main features of a compiler that we constructed for translating the μFOOPS algebraic specifications in C++ and Eiffel and the conclusions resulted using this language and compiler in specifying some applications. The paper is structured in four sections.

The first section mentions the problem leading to our idea: the join of formal and heuristic object-oriented analysis and design methods. For this end to be reached we should start with the automatic code generation from algebraic specifications.

In the second section, named *The specification language μFOOPS*, we describe our specification language. We have chosen to mention both the main FOOPS concepts retained in our language and the restrictions that we imposed over the specifications in order to be accepted by our compiler. We relate our implementation with the one mentioned in [8].

The third section, named *The Compiler*, consists of two parts. The first one mentions the semantic checking performed by the compiler. We give extra information only for the checkings that modify the standard. The second one presents what exactly is generated from the specifications, whith an example (the specification of lists and their translation in C++).

In the last section we present the conclusions drawn from using the compiler: the quality of the generated code, the extent in which the language can be used for specifying real applications. We also mention two different problems that we tryed and succeded to solve using our language (the monitorisation of a Home Heating System and a clasic backtracking algorithm), and the future work directions.

Received by the editors: September 29, 1996.

1991 *Mathematics Subject Classification.* 68Q45, 68Q52, 68Q60, 68Q65, 68N20.

1991 *CR Categories and Descriptors.* D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.4 [Software Engineering]: Program Verification – assertion checkers, correctness proofs, validation; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs – specification techniques.

1. Introduction

The development of certain object-oriented analysis and design methods that integrate formal techniques into heuristic methods enjoys an increasing attention from the research community nowadays.

Each of these two kinds of methods has its advantages and disadvantages. The diagrams resulted from applying heuristic methods are self-evident and intuitive, whiles the formal specifications offer support for rigorous description and behavioral checking in an algebraic framework. One of the problems still unsolved consists in combining these two methods in order to obtain a maximum benefit.

Automated code generation and behavioral properties checking are two fields where formal methods are with no doubt superior, and the chances that they will be integrated in heuristic methods are good.

In the following sections we will present the choices that we made and the results that we achieved in constructing a code generator for an algebraic specification language. For the beginning our purpose was to define or find a language as simple as it ca be without loosing its power of expression, and which uses concepts that have a direct mapping in Object-Oriented Languages (OOL).

2. The specification language μFOOPS

In defining the specification language our aims were to allow a natural expression of the information obtained during the analysis of a problem as well as to offer support for formalreasoning and automated code generation. After studying a large set of existent specification languages, we decided to go for a subset of FOOPS (Functional and Object-Oriented Programming System) called μFOOPS that partially satisfies our goals.

FOOPS is a very high level object-oriented specification language with an executable subset. It has Abstract Data Types (ADTs), classes, objects, overloading, polymorphism, inheritance and many other facilities non-existent in nowadays programming languages such as parametrised modules, module interconnection, mixfix syntax for operators. FOOPS was developed as an extension of OBJ, a functional specification language. It is the result of unifying Functional and Object-Oriented Programming (OOP), and it was first described in [5]. A complete description of the language can be found in [8]. In the following paragraphs we will present some of the concepts used by FOOPS and by our language and the restrictions that we imposed over FOOPS specifications.

The FOOPS type system makes two important distinctions. On the one hand data are not objects. Data are characterized by a state that cannot be damaged. Numbers and colors are for instance data elements. Objects have an internal state that may evolve with time, for instance a car or a CRT is an object. When these two concepts are merged, as in many programming languages, it is likely to run into confusions. As a consequence in FOOPS data elements are collected in sorts and objects in classes. An ADT in FOOPS is formed by a sort and some functions associated to it. Functions may take objects or data as arguments and return an object or a data element.

On the other hand classes are not modules. Object-Oriented Programming Languages (OOL) usually consider the syntactic structure for defining a class and its associated attributes and methods as the main programming unit. This is not the case with FOOPS, where the module is the main programming unit, allowing the programmer to define together the related classes, sorts and operations.

Having got the distinction between sorts and classes we will need to make a difference between the two levels of a specification: the functional and the object level. At each level there exist two kinds of modules: the ones that encapsulate executable code and the others that declare properties. The former are named **modules**, the latter are named **theories** (object or functional). The compiler described in this paper takes a μFOOPS specification and translate it in C++ or Eiffel. **Theories** are used for checking the behavior of the described entities and for a highly flexible mechanism of module parametrisation, and they were not included in our purposes for the moment.

The functional module is the main programming unit at the functional level that encapsulates executable code. A functional module defines one or more ADTs, consisting of data sets and operations defined using them. A data set is called a **sort** and the operations are called **functions.** μFOOPS allows only functions in prefix notation. Mixfix notation would increase too much the difficulty of the syntax analysis process and would generate problems at translation time due to the lack of correspondence in usual OOLs. The absence of mixfix notation does not affect the power of expression of μFOOPS.

The result generated by a function is described by axioms. Axioms are term equalities that may or may not be conditioned by a Boolean valued term. An axiom at this level has the following form:

[c]**ax** <Term> = <Term> [**if** <Term>]

Following is the functional module that describes BOOLEAN values used by μFOOPS:

```
fmod BOOLEAN is
    *** BOOLEAN sort declaration
    sort Boolean.
    *** The functions defined for this sort
    fn True : -> Boolean.
    fn False : -> Boolean.
    fn Not : Boolean -> Boolean.
    fn And : Boolean Boolean -> Boolean.
    fn Or : Boolean Boolean -> Boolean
    *** A BOOLEAN variable
    var x : Boolean.
    *** The description of the behavior
    ax Not(True()) = False().
    ax Not(False()) = True().
    ax And(True(),x) = x.
    ax And(False(),x) = False().
    ax Or(True(),x) = True().
    ax Or(False(),x) = x.
endf
```

REMARK: From the point of view of the code generation process, functional modules are only used to specify basic types, that are usually predefined in OOLs. They are hard to implement automatedly in an OOL because of this tight connection to the predefined types. Thus, for sorts and functions there will have to be written manual implementations in the target language. Our compiler will build only the declarations for these entities, the actual code that would have had to be deduced from the axioms being ignored. Having in mind that we will use sorts only for basic types, we renounced to the possibility of declaring inheritance between sorts.

The object module is the main programming unit at the object level that encapsulates executable code. An object module may define one or more classes, which are potential collections of objects. The attributes and methods associated to a class describe the internal structure and the behavior of the objects of that class. An object module may as well contain descriptions of ADTs similar to those described above, at the functional modules.

The properties of the attributes and methods are specified in terms of axioms too. The axioms may be conditional or unconditional and the terms involved may contain references to functions, methods, attributes and variables.

REMARK: Our compiler uses object level specifications in order to obtain actual code for classes, attributes and methods in typed OOLs. Object level entities have a natural correspondent in OOLs and we were able to obtain efficient code for them. Of course, there still exist some restrictions at this level concerning the forms of the axioms and the signature of the methods, restrictions imposed by the standard FOOPS and the Oxford implementation [8] too.

Another aspect worth to be mentioned is the possibility to import modules (object or functional) in other modules. By using these mechanisms we can obtain module hierarchies, module importation being called module inheritance by the authors of FOOPS.

3. The compiler

In the following sections we will present some of the aspects considered to be relevant, concerning the semantic checking and the code generation (in C++ or Eiffel) in our compiler. We will use as an example the specifications for lists.

3.1. Semantic Checking.

After the syntactic analysis of the specifications, the compiler does some semantic validity checking concerning the issues mentioned below. We will describe only the changes made with respect to standard FOOPS.

1. **Module importation.** A module can import other modules both at the functional and the object level. Although FOOPS allows three kinds of module importation (protecting, extending and using), μFOOPS offers support only for using. The other two possibilities impose restrictions over the use of the imported module, which are not hard to check but have no correspondent in usual OOLs.
2. **Inheritance relationship.** FOOPS allows inheritance for both classes and sorts. μFOOPS offers inheritance only for classes, for sorts being considered unimportant. The language does not allow direct repeated inheritance.
3. **Variable declarations.**
4. **Operations signatures.** We do not impose any restriction over the names used for the operations at compiling time. However, the names will not change in the generated code, so they should not conflict with predefined or library names in the target language, nor should they conflict among themselves.
5. **Redefinitions of attributes and methods.** For an attribute or a method, the name indicates that that operation is redefined if it is the same one as the ancestor attribute or method name. Redefinitions must obey the covariance

rule for parameters' type. However, when translating in C++, the parameters' type change will not reflect in the generated code because of the lack of support for this in the target language.

6. **Type of the terms.**
7. **Axioms.** At axiom checking stage we get rid of the axioms that are not in one of the recognized forms (see next section). These forms are those accepted by the Oxford implementation, and they proved to be sufficient for describing the behavior of objects appearing in a great variety of applications.

3.2. **The code generation process.** We will present in the next paragraphs what exactly is generated from the object level specifications, with examples for the specification of lists.

Besides the sorts INTEGER and BOOLEAN we will need the specifications for the classes LIST and OBJECT, shown below. The specifications are not quite identical to those well known by the FOOPS community (see for instance the axioms for AddHead), but the changes are small and justified.

```
omod OBJECT is
    class Object.
endo
omod LIST is
    using OBJECT.
    using INTEGER.
    using BOOLEAN.
    subclass List < Object.
    at head : List -> Object.
    at tail : List -> List.
    at empty : List -> Boolean
                [default:(True())].
    at IsEmpty : List -> Boolean.
    at GetHead : List -> Object.
    at GetTail : List -> List.
    at GetCount : List -> Integer.
    me AddHead : List Object -> List.
    me AddTail : List Object -> List.
    me RemoveHead: List -> List.
    me RemoveTail: List -> List.
    me RemoveAll : List -> List.
    var L : List.
    var O : Object.
    var I : Integer.
```

```
*** IsEmpty —
    ax IsEmpty(L)=empty(L).
*** GetHead —
    ax GetHead(L)=head(L).
*** GetTail —
    ax GetTail(L)=tail(L).
*** GetCount —
    cax GetCount(L)=Zero()
    if IsEmpty(L).
    ax GetCount(L)=
        Succ(GetCount(tail(L))).
*** AddHead —
    ax empty(AddHead(L,O))=False().
    ax head(AddHead(L,O))=O.
    ax tail(AddHead(L,O))=L.
*** AddTail —
    cax AddTail(L,O)=AddHead(L,O)
    if IsEmpty(L).
    ax AddTail(L,O)=AddTail(tail(L),O).
*** RemoveHead —
    cax head(RemoveHead(L))=
            head(tail(L))
    if Not(IsEmpty(L)).
    cax tail(RemoveHead(L))=
            tail(tail(L))
    if Not(IsEmpty(L)).
    cax empty(RemoveHead(L))=
            empty(tail(L))
    if Not(IsEmpty(L)).
*** RemoveTail —
    cax RemoveTail(L)=RemoveHead(L)
    if And(Not(IsEmpty(L)),
            IsEmpty(tail(L))).
    cax RemoveTail(L)=
            RemoveTail(tail(L))
    if And(Not(IsEmpty(L)),
            Not(IsEmpty(tail(L)))).
*** RemoveAll —
    cax RemoveAll(L)=RemoveHead(L);
            RemoveAll(L)
    if Not(IsEmpty(L)).
endo
```

3.2.1. *The modules.* The implementation of modules can be viewed at least from two different perspectives:

- We may consider the module as a unit for structuring the specifications, which reflects only in a small measure in the generated code. This strategy was adopted for dealing with modules when generating C++ code. We used modules only to structure the code in files. Thus the sorts, classes and operations specified in one module will be declared and implemented in one file.
- We may as well consider the module as a unit for structuring the specifications that reflects in the generated code. The modules will be implemented by classes, the import relationship will be implemented by the class inheritance. This strategy was adopted for the Eiffel code generation.

3.2.2. *The Types.* As we mentioned above, we did not implement the sorts and the functions, but only generated declarations for them. However, there still were some language-dependent choices to make:

- In C++ the sorts are automatedly implemented by the type **int** (which does not mean that one cannot modify that by hand) in the file corresponding to the module which declared the sort.
- In Eiffel the sorts are implemented by classes without features that inherit the class corresponding to the module in which sort was declared.

The classes are implemented by classes in the target language. The members (features) of these classes will be the attributes and the methods described in the specifications. The inheritance at the specification level is implemented by inheritance in the target language. There exist the following language-dependent differences:

- In C++ the class declaration resides in the declaration file of the module in which the class was defined. The implementation of the methods and calculated attributes are in the implementation file of the module.
- In Eiffel a class inherits the class corresponding to the module in which it was defined, in order to be able to use the functions and Entry Time Objects (ETOs) declared in the module. Every parent class is inherited twice in order to use its creation feature.

3.2.3. *The operations.* We will discuss the four kinds of operations (functions, attributes, methods and ETOs) separately as each of them raises specific problems.

Function declaration does not raise problems. For each function we build a function declaration with the same name and types for parameters (C++). Special cases are the languages that do not have a concept of function (is Eiffel for instance). In these cases functions are considered features in the class corresponding to the module in which they were declared.

Stored attributes are implemented by fields (instance variables) in their classes. Their default values (specified by the **default** clause) are considered when generating code for the constructor (creation feature) of the class.

Derived attributes are implemented by query methods (features) in their classes, they return a value that depends on the state of the object but not modify that state.

Methods are implemented by methods in the target language that modify the state of the object upon which they are called. They can directly modify the values of the attributes or call other methods, depending on the type of the axioms that describe their behavior (Direct Method Axioms / Indirect Method Axioms). In order to satisfy the semantics of the specifications, we decided to include two features for each method when generating Eiffel code, one for implementation which does the job and returns the (receptor) object, and another one for interface, which calls the former one and does not return anything. The operations for which we generate actual code (not signatures) are: constructors, derived attributes, methods and ETOs.

Constructors are generated based on the default values of the stored attributes. For each attribute that has a default value a line is generated in the constructor to initialize that attribute. The creation feature in Eiffel is named **make**.

Code for derived attributes is obtained using the list of axioms attached to each attribute. In the most general case that list may contain some conditional axioms followed or not by an unconditional axiom (the list is made like this during the axiom processing that follows semantic checking). For instance, if the attribute is defined by the following axioms (the order is important):

```
cax attr(...) = <Term_1>
    if <Cond_1>.
  ...
cax attr(...) = <Term_n>
    if <Cond_n>.
ax attr(...) = <Term_n+1>.
```

then the generated code looks like this:

```
if Cond_1 then
    return Term_1;
...
if Cond_n then
    return Term_n;
return Term_n+1;
```

The code for the methods specified by DMAs will contain a list of assignments of values to attributes. For each attribute we have a list of axioms that describe the effect of applying the method over that describe the effect of applying the method over that attribute in certain conditions. For instance, for an attribute and a method we may have a list of axioms of the following form:

```
cax at(me(...)) = <Term_1>
    if <Cond_1>.
...
cax at(me(...)) = <Term_n>
    if <Cond_n>.
ax at(me(...)) = <Term_n+1>.
```

Then the generated code will look like this:

```
if Cond_1 then
    at := Term_1;
...
elseif Cond_n then
    at:= Term_n;
else
    at := Term_n+1;
```

The code for the method will contain such a sequence for each attribute that has a value specified. At the beginning of the method we make a copy of the object because we might need the old values of the attributes after they were modified. This copy is deallocated if it has not been used.

The code for the methods specified by IMAs will contain a sequence of calls of other methods if the axioms are:

```
cax meth( ... ) =
    <Term_1_1>;<Term_1_2>; ...
    if <Cond_1>.
...
cax meth( ... ) =
    <Term_n_1>; <Term_n_2>; ...
    if <Cond_n>.
ax meth( ... ) =
    <Term_n+1_1>; <Term_n+1_2>;
    ...
```

the generated code will be:

```
if Cond_1 then
    Term_1_1;
    Term_1_2;
    ...
    return
...
if Cond_n then
    Term_n_1;
    Term_n_2;
    ...
    return;
Term_n+1_1;
Term_n+1_2;
...
return;
```

ETOs are implemented like the functions, but they always return the same result. The first time we call an ETO an object is created, initialized according to the DMAs of the ETO and then returned. The next calls to the ETO will return the same object.

For the lists, the C++ code generated by the compiler is presented in Appendix.

4. Conclusions and future work

This paper presented a compiling technique for translating a class of algebraic specifications in two typed OOLs (C++ and Eiffel). This techniques has been implemented in a compiler for μFOOPS with versions for Windows, DOS and UNIX (Posix). The compiler has been used to translate the example mentioned in the previous section as well as other examples, some of them mentioned below.

As we mentioned from the beginning, our idea was to reach to those aspects from the formal and heuristic techniques that may lead to a worthy combined approach. We considered first the code generation process because it's well known the fact that without a rigorous description of the behavior, the generated code is mostly signatures.

The code generated by our compiler is efficient and is obtained in a very short time. Due to its readability it can be easily refined by hand when wanted. The example presented above does not lead to any conclusions about the usage of the compiler in real applications. It was chosen because it is well known and the code is easy to understand. We used μFOOPS to specify a Home Heating System. For this example we used the results obtained from the analysis as they were presented in [2]. Our conclusion was that there is a mapping between the entities in the heuristic model's diagrams and some of the concepts manipulated by μFOOPS, and the translation from diagrams to specification is natural.

We used the language and the compiler to specify problems that contain a lower degree of declaratives, for instance some classic algorithms. The implementation of a backtracking for the Queens Problem was not a hard work.

One of the directions towards which we intend to lead our work is the possibility to generate specifications from the diagrams in heuristic models. We also intend to compare the results obtained this way with the results obtained by using other hybrid methods (like Syntropy).

The uses that we can find for our compiler give us reasons to extend it in at least two directions that were neglected till now: parametrisation and specification of concurrency.

References

[1] R. Breu, *Algebraic Specifications in Object-Oriented Programming Environments*, Springer Verlag, 1992.

[2] G. Booch, *Object Oriented Design with Applications*, The Benjamin / Cummings Publishing Company Inc., 1991.

[3] L.M.G. Fejs, H.B.M. Jonkers, *Formal Specification and Design*, Cambridge University Press, 1992.

[4] N.E. Fucs, *Specifications Are (Preferably) Executable*, Software Engineering Journal, September 1992.

[5] J. Goguen, J. Mesenguer, *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, in B. Shriver and P. Wegner editors, Research Directions in Object-Oriented Programming, M.I.T. Press, 1987, pp.417 - 477.

[6] K. Lano, H. Haughton, *Object-Oriented Specification Case Studies*, Prentice Hall International, 1994.

[7] B. Meyer, *Eiffel, The Language*, Prentice Hall International, 1992.

[8] L. Rapanotii, A. Socorro, *Introducing FOOPS*, Technical Report, Oxford University, Research Computing Laboratory 1992.

[9] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

[10] P. Turlier, *La compilation des types abstraits algebriques du langage LOTOS*, These de doctorat, Conservatoire National des Arts et Metiers, Grenoble 1993.

Appendix

```
/* LIST module interface */
#ifndef _LIST_h_
#define _LIST_h_
#include "OBJECT.h"
#include "BOOLEAN.h"
#include "INTEGER.h"
/* classes */
     class List;
class List : public Object {
protected:
     Object* head;
     List* tail; Boolean empty; public:
     List();
     virtual Boolean IsEmpty();
     virtual Object* GetHead();
     virtual List* GetTail();
     virtual Integer GetCount();
     virtual void AddHead(Object* p1);
     virtual void AddTail(Object* p1);
     virtual void RemoveHead();
     virtual void RemoveTail();
     virtual void RemoveAll();
     };
/* functions */
#endif
/* LIST module implementation */
#include <stdio.h>
#include <memory.h>
#include "LIST.h"
List::List(){
     head=NULL;  tail=NULL;
     empty=::True(); }
Boolean List::IsEmpty(){
     return (this)->empty;
     return (Boolean)0;}
Object* List::GetHead(){
     return (this)->head;
     return (Object*)0;}
List* List::GetTail(){
     return (this)->tail;
     return (List*)0; }
Integer List::GetCount(){
     if ((this)->IsEmpty())
          return ::Zero();
     return ::Succ(((this)->tail)->GetCount());
     return (Integer)0; }
void List::AddHead(Object* p1){
     List* tmp=new List;
     memcpy(tmp,this,sizeof(List));
     empty=::False();
     head=p1; tail=tmp;}
void List::AddTail(Object* p1){
     if ((this)->IsEmpty()){
          (this)->AddHead(p1);
          return;}
     ((this)->tail)->AddTail(p1);}
void List::RemoveHead(){
     List* tmp=new List;
     memcpy(tmp,this,sizeof(List));
     if (::Not((tmp)->IsEmpty()))
          head=((tmp)->tail)->head;
     if (::Not((tmp)->IsEmpty()))
          tail=((tmp)->tail)->tail;
     if (::Not((tmp)->IsEmpty()))
          empty=((tmp)->tail)->empty;
     delete tmp;}
void List::RemoveTail(){
     if (::And(::Not((this)->IsEmpty()),
               ((this)->tail)->IsEmpty())){
          (this)->RemoveHead();
          return;}
     if (::And(::Not((this)->IsEmpty()),
               ::Not(((this)->tail)->IsEmpty()))){
          ((this)->tail)->RemoveTail();
          return;} }
void List::RemoveAll(){
     if (::Not((this)->IsEmpty())){
          (this)->RemoveHead();
          (this)->RemoveAll();
          return; }
```

"Babeş-Bolyai" University, Research Laboratory on Computer Science, Cluj-Napoca, Romania

E-mail address: {dorel,chiorean,iulian}@cs.ubbcluj.ro