Using Object-Oriented Languages to Implement Algebraic Specifications

Hermano Perrelli de Moura*
Luis Carlos de Sousa Menezes**

Universidade Federal de Pernambuco, Pernambuco - Brazil

Abstract. Some concepts defined in algebraic specifications are similar to some concepts of the object-oriented paradigm, for example the sorts of algebras is similar to classes in object-oriented programs. This paper identify these similarities and explore them to define a method that implements algebraic specifications converting the entities defined in the specification to their object-oriented equivalents.

1 Introduction

In this paper we present the results of a study that compares the concepts of algebraic specifications languages like OBJ [GKK+88] and object-oriented languages like C++ [Str91]. This study will produce the definition of a translation method that implements algebraic specifications mapping the defined entities to their object-oriented equivalent forms.

2 Algebraic Specifications

An algebraic specification defines a set of entities named sorts and operators that receive arguments of defined sorts and produce a values of a specified sort. In this section we introduce the of notation of order-sorted algebra described in [Gogar] that will be used in this paper.

2.1 Signatures

A signature is used to identify the sorts and operators defined in the specification. A order-sorted signature is represented by the tuple (S,$\leq$,Σ), where:

- S is the set of the sorts defined by the specification;
- $\leq$ represents the inclusion relation between the sorts defined in the specification. If $x \leq y$ we say that the sort x is a subsort of the sort y and every element of the sort x also belongs to the sort y.
- Σ represents the operations defined by the specification and is defined by the expression: $\{\Sigma_{w,s} \mid w \in S^{*} e s \in S\}$, where $\Sigma_{w,s}$ denote the set of operators that receives arguments of sorts w and produces elements of sort s.

* E-mail: hermano@di.ufpe.br
** E-mail: lcsm@di.ufpe.br

The operator $o \in \Sigma_{[],s}$ where: [] denots the empty sequence, means that this operator receives no arguments. This kind of operator represents the constants symbols of the sort s.

An operator o is overloaded iff $o \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ for some $(w, s) \neq (w', s')$. The translation method described in the next sections will not consider overloaded operators however it can be extended to be able to process them.

2.2 Terms

The set of the valid terms defined by the signature $(S,\leq,\Sigma)$ is given by the expression: $\{\mathcal{T}_{\Sigma,s} \mid s \in S\}$, where $\mathcal{T}_{\Sigma,s}$ is the set of the terms of the sort s and is obtained using these rules:

- $\Sigma_{[],s} \subset \mathcal{T}_{\Sigma,s}$
- $\mathcal{T}_{\Sigma,s'} \subset \mathcal{T}_{\Sigma,s}$ if $s' \leq s$.
- If $\sigma \in \Sigma_{[s_1,\ldots,s_n],s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}$ then $\sigma(t_1, ..., t_n) \in \mathcal{T}_{\Sigma,s}$
- If x is a variable of the sort s, $x \in \mathcal{T}_{\Sigma,s}$.

2.3 Equations

Equations in a order-sorted specification are used to define the semantic of the operators. For a signature $(S, \leq, \Sigma)$ a equation is formed by the tuple (X, t_1, t_2), where:

- X if a set of variables names.
- t_1 and t_2 are valid term in the signatures

the notation used to express this equation is:

$\forall X, t_1 = t_2$

Another kind of equation is the conditional equations, formed by the tuple (X, t_1, t_2, C), where the condition C is formed by a set of equations. The notation used to express this kind of equation is:

$\forall X, t_1 = t_2 \text{ if } C$

2.4 Example

The Figure 1 shows an example of algebraic specification. It was writen using the formal language OBJ3 [GKK+88] and shows the specification of the natural numbers. This specification has the following properties:

- The line 1 declares the sorts defined in the specification.
- The lines 2-5 define the operators of the specification. The operator `0`, line 2, receives no params and produces a natural as results, this notation is used to represent the constants of specification. The operator `succ`, line 3, is used to represent the successor of a natural. The operator `sum`, line 4, calculates the sum of two given naturals. Finally, the operator `fib` that calcules the number of fibonacci sequence.

```
(1)  sorts natural
(2)  0 : → natural
(3)  succ _ : natural → natural
(4)  sum _ _ : natural, natural → natural
(5)  fib _ : natural → natural
(6)  var x, y : natural
(7)  sum 0 x = x
(8)  sum (succ x) y = succ (sum x y)
(9)  fib 0 = 0
(10) fib (succ 0) = succ 0
(11) fib (succ succ x) = sum (fib (succ x)) (fib x)
```

Fig. 1. Natural Specification

- The line 6 defines the variables symbols used by the equations.
- The lines 7-11 represent the equations of the specification.

3 Similarities Between Order-Sorted Algebras and Object-Oriented Programs

Comparing the concepts existing in order-sorted algebras and object-oriented programs, we have noted the existence of some similarities between the structures of these models. Basically, we have identified the following similarities:

- The concept of sorts is similar to the concept of classes in the Object-oriented paradigm.
- The sort inclusion relationship is similar to the class inheritance relationship.
- Sorts' elements are similar to objects.
- Sorts' operators are similar to methods of classes.

The study of these similarities was the initial step in the development of a method to translate order-sorted specifications into object-oriented programs. This method will be described in the next section.

4 Translation of Algebraic Specifications

Using the similarities described in the last section we propose a method that implements order sorted algebras specifications. The method that will be showed here accepts an algebraic specification source and produce an equivalent class library. This class library will be able to:

- Represent the terms specified by the source specification like object-oriented expressions.
- When this expression is executed, the library is able to apply the rewriting equations [DJ90], specified in the source specification, producing a object that represents the normal forms terms.

The next subsections will show the guidelines used by this method to generate the class library.

4.1 Class Hierarchy

When an algebraic specification of signature $(S,\leq,\Sigma)$ is translated to a object-oriented program, the target program will have the following class hierarchy:

1. There will be a class named Sort that will be a superclass of all classes generated in this process.
2. Each sort $x \in S$ will have a class, called $C(x)$, destined to represent the elements of the sort x.
3. For all classes x and y, if $x \leq y$ then the class $C(x)$ will be, directly or indirectly, a subclass of the class $C(y)$.
4. All non constant operation symbol $op \in \Sigma_{[p_1,\ldots,p_n],s}$ will have in the target program a class named his operator class, represented by $C(op, p_1, ..., p_n)$. This class is a subclass of $C(s)$ and will be used to represent the terms produced by the operator op who can't be reduced using the rewriting equations. Operator classes also have instance variables to represent the argument's values and a constructor method to initialize the instance variables.

4.2 Initial Objects

During the translation of an specification, every constant operator $o \in \Sigma_{[],s}$ will be implemented like a new object of the class $C(s)$. The initial value for an object built by the translation of the operator $o \in \Sigma_{[],s}$ is defined using these rules:

- If the source specification have a equation in the form: $o = t$, then the initial value of the object generated by this operator is the expression produced by the translation of the term t like described in the Section 4.5
- Otherwise, the initial value of this object is a new instance of the class $C(s)$, produced by the translation of the sort s.

4.3 Pre-defined Methods

Every class produced by the translation method will respond the following pre-defined methods:

- `Equal` : Returns `true` if two objects represent the same term.
- For all operator $op \in \Sigma_{[p_1,\ldots,p_n],s}$ is defined a method named `Match_`op used to verify whether the object is a term built with this operation.

4.4 The Operator Methods

Every operator $op \in \Sigma_{[p_1,\dots,p_n],s}$ will be associated with a new method called his operator method. This method is defined in the class that implements the first argument ($C(p_1)$), receives objects that represent the operator's other arguments classes and returns an object of the $C(s)$ class. The functionality of an operator method is defined by the rewriting equations existing in the source specification.

The body of this operator method contains the code segments generated by equations of the form:

$$op(t_1, ..., t_n) = t \text{ if } C$$

each one of those equations will produce a code segment that:

1. Will test if the term represented by the call of operator method match with the specified by the terms t_1,...,t_n and the condition C. This test will use the pre-defined methods `Equal` and `Match`.
2. If this test is satisfied, the value of the term t is calculated, using the rules described in the Section 4.5 and the object resulting is returned as the method result.

To exemplify this process, the code generated by the equation:

$$\mathsf{sum}\ (\mathsf{succ}\ x)\ y = \mathsf{succ}\ (\mathsf{sum}\ x\ y)$$

is:

```
{
  natural local0;
  if (p1->Match_succ(local0))
  return local0->sum(p2)->succ();
}
```

The `if` command will test if the first argument is constructed by the operator succ using the `Match_succ` method. This method returns true or false and accepts a local variable that will be assigned with the argument of the succ operator if the test succeeds. If the test results the true value the `return` command is executed so this command evaluate the right term and return it as the method result.

If no code segment generated by this kind of equations produce the result of the method call, the last command of the method's body will create a new object of the operator class generated by the operator *op* and return it as the method result.

4.5 Representing Terms

The class library generated by the translation of an algebraic specification, is able to represent terms like object-oriented expressions. If the map m returns the expression who translates the variable symbols, the object-oriented expression who represents a term t, denoted be the expression $\mathsf{evalue}(t, m)$, is produced using this rules:

– If $t \in \Sigma_{[],s}$, the expression that represents it is the variable produced by t during the translation of this operator.
– If $t = op(a_1, a_2, ..., a_n)$, where $op \in \Sigma_{[s_1,...,s_n],s}$, the object-oriented expression that represents it is: "r_1->$mo(r_2, \ldots, r_n)$", where:
  - om is the operator method produced by the operator op during the translation of the source specification.
  - $r_x = \mathsf{evalue}(a_x, i)$, $1 \leq x \leq n$
– If t is a variable symbol, the expression produced by the operator is given by the following sentence: $m(t)$.

5 A Small Example

In this section, we show the object-oriented code produced by the specification showed in the Figure 2. That specification, defines the sort stack that represents stacks of elements belonging of the sort item. The sort ne-stack represents the sort of the no empty stack and it is a subsort of stack, line 1.
The sort stack has a constant represented by the operator empty-stack, defined in the line 3, and the sort item has the constants a, b and c, defined in the line 6.
The operators push and pop, defined in the lines 4-5, are used to operate stacks. The semantic of these operators are defined by the equation of the last line.

(1) sorts stack, ne-stack , item
(2) subsorts stack, ne-stack
(3) empty-stack : $\rightarrow$ stack
(4) push _ _ : item, stack $\rightarrow$ ne-stack
(5) pop _ : ne-stack $\rightarrow$ stack
(6) a, b ,c : $\rightarrow$ item
(7) var x : item
(8) var y : stack
(9) pop (push x y) = y

Fig. 2. Stack Specification

Implementing the source specification, we obtain the object-oriented program showed bellow:

```
class Sort
{
  int Equal(Sort s) {return this==s;}
  int Match_push(item &i,stack &c) {return 0;}
```

```
};

class item : public Sort
{
  stack Push(stack p) {return new push(this,p);}
};

item a = new item;
item b = new item;
item c = new item;

class stack : public Sort
{
};

stack empty-stack = new stack;

class ne-stack : public stack
{
  stack pop()
  {
    item i;
    stack s;
    if (Match_push(i,s)) return s;
    return new pop(this);
  }
};

class push : public ne-stack
{
  item i;
  stack s;
  push(item p1,stack p2) { i = p1;s = p2; }
  int Equal(Sort so)
    {
      item i2;stack s2;
      if (so->Match_push(i2,s2))
        return s->Equal(s2) && i->Equal(i2);
    }
  int Match_push(item &p1,stack &p2)
    {
      p1=i;p2=s;
      return 1;
    }
};

class pop : public stack
{
};
```

That will have the following properties:

- The class `Sort`, defined in the lines 1-5, is superclass of every class defined in this process.
- The sorts defined by the specification: item, stack and ne-stack; will be translated as the classes defined in the lines 7, 16, 22.
- The class `ne-stack` is a subclass of the class `stack` because the sort ne-stack is a subsort of the sort stack.
- The constants operators will be implemented like new objects that are showed in the lines 12,13,14 and 20.
- The operator push and pop will produce class operators, lines 33-53, to represent the terms built by them.
- The operator push will produce the method operator showed in the line 9. As there is no equation that rewrites terms generated by this operator, the only functionality of the operator method is produce objects of the operator classes.
- The operator pop will produce the method operator showed in the lines 24-30. This method will test the rewriting equation showed in the line 9 of the source specification and will produce objects of the operator classes whether the test fails.
- The code generated to the pre-defined method `Equal` is showed in the lines 3 and 38-43.
- The code generated to the pre-defined method `Match_push` is showed in the lines 4 and 44-48

6 Conclusions and Related Works

The method proposed in this paper is able to implement efficiently algebraic specifications. The main drawback of this method is the lacking of a formal verification of the method's soundness whose definition still are in progress.

An implementation of this method is almost ready for distribution and will be available in the web site of the RAT Project [Rat].

The main motivation for the definition of this method was support a of compiler generation method, defined in [Men98], that produces automatic implementations of action semantics descriptions [Mos92, Wat91] to improve the systems that use fixed implementations.

References

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, chapter 6. Elsevier Science Publishers, 1990.

[GKK+88] Joseph Goguen, Claude Kirchner, Hélèno Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to obj3. In *Conference on Conditional Term Rewriting*, volume 308 of *Lecture Notes in Computer Science*, Springer 1988.

[Gogar] Joseph A. Goguen. *Theorem Proving and Algebra.* To appear.

[Men98] Luis Carlos de Sousa Menezes. Uso de orientação a objetos na prototipação de semântica de ações. Master's thesis, Universidade Federal de Pernambuco, 1998.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Rat] Homepage of the RAT Project. http://www.di.ufpe.br/~rat.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice Hall, 1991.