Anul I 1996

STUDIA
UNIVERSITATIS "BABEŞ-BOLYAI"
INFORMATICA
1

Redacţia: 3400 Cluj-Napoca, str. M. Kogalniceanu nr. 1 • Telefon: 194315

SUMAR – CONTENTS – SOMMAIRE

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

NEVERTHELESS, THERE IS A COMPUTER SCIENCE

E. MUNTEANU AND M. FRENŢIU

Is there a Computer Science, and if there exists this Science, what is it? What are its fields, and how must we teach it?

The debate on this subject is an old one [1, 2, 4, 5, 6, 7, 8, 10, 11]. There are today many Computer Science Departments all over the world, there are many specialized Computer Science Journals, and there are a lot of applications. Now, there isn't a human activity where this new science is not present. In spite of all these facts there are our colleagues from Mathematical Department who are saying that there is not a Computer Science, that it emerged only a new branch of Mathematics. It is not our intention here to polemize with them on this subject. But this new Journal is a new fact that proves the existence of Computer Scientists in our Department.

Therefore, among the other Journals of our University, a new Scientific Journal appears. The Computer Science Department of our University will support an Academic Journal of Science and Information Technology. This journal addresses to scientific Community and tries to present the scientific results of this field.

We all know the fast evolution of this Computer Science. There was not in the entire history of all sciences such a high rate of development as Computer Science had. All happened in only 50 years. And in the last 25 years the computers power grew up 1000 times; it is presumed that in the next 25 years it will again grow up 1000 times.

A journal exists for its readers. The best journal is that one that satisfies these readers. Computer Science is not only very dynamic, but also an immense field; from Mathematical Foundations of Computers to Computer Applications, from Formal and

Received by the editors: July 12, 1996.

Programming Languages Theory to Software Engineering, from Data Bases to Computer Architecture, from Operating Systems to Artificial Intelligence, from Computer Networks to Communications Management, and these are only a few. For the beginning, research in all fields is welcome in the pages of this journal. We accept any cooperation, we wait for all computer scientists to become not only our readers, but also, to publish their research in the pages of this journal.

Publishing the most recent results of scientific research, this journal will give us the possibility to be informed and to understand, to spread our results, or to show new research directions, even if we are at the beginning and the main research and computer technology are created in the more developed countries. But the important ideas can appear everywhere, and the change of views is very important to us.

Studia Journal, Computer Science series, will outline our scientific standing in the Romanian and world's academic community. The detachement of this journal from the mathematical one is a very important step made by us.

A few years ago, in a didactic activity with school teachers, the second author of this paper has compared Computer Science and Mathematics with two brothers, first - very young and second - a mature one. The mathematicians disagreed, but the situation is rapidly changing. Computer Science is becoming a very fast maturing science, and no other science has appeared in such a short time. More, Computer Science is spread over almost all human activities. It has influenced and helped the development of all other sciences. It made possible to solve some problems impossible to be solved in the absence of computers.

The history and the progress made in Computer and Information Technology is presented in a special issue of IBM Journal of Research and Development [12]. Then, the history of Computer Science Education can be read from [2, 6, 8].

But what is Computer Science? Certainly, Computer Science is not the science of computers. It appeared due to these wonderful machines called computers. But as Hartmanis defined it [7], it is the science of information processing. It deals with the

systems that create, store, and process information. It differs from all other sciences by its immense field of applications.

Some similarities and differences between Mathematics and Computer Science are presented by Knuth [9]. Certainly, Computer Science has emphasized the important role of constructions in all branches of Mathematics. One important difference is mentioned by Knuth in [9]: Computer Science is a very young science which is developing very fast. The curriculum is also very rapidly changing, in contrast to a much slower change present in mathematical courses. It has in common with Mathematics the process of abstractions. The computer scientists has to create complicate systems, to design them at different levels of abstraction, to implement them with high precision, to motivate their correctness.

Certainly, Mathematics and Computer Science influence each other [1, 9, 10]. The first one offers its methods, the second one puts a lot of new problems not only for computer scientists but also for mathematicians.

The first generation of Computer Science undergraduates of our University started their studies in 1971. Therefore, Computer Science activity at our University is around 25 years old. During these years the research activity covered the following fields: programming and formal languages, programming methodology, fuzzy systems and their applications, artificial intelligence, parallel computation and distributed systems, operating systems, data bases, computer graphics, computer applications, and teaching methodologies. This activity is described in [3]. Here is a short survey of this activity.

In the field of programming and formal languages, as everywhere in the world at that time, there were studies in syntactic and semantic analysis and translators: interpreters and compilers with error correcting capabilities.

Then, we were concerned with the improvement of the programming methods and languages, due to the necessity of the teaching process and, also, due to the existing

software crisis. There were studies related to structured programming, improving programming languages, the methodology of programming, object- oriented programming, program correctness, program analysis.

The mathematical foundation of fuzzy systems and their applications were also studied. The most important results are: the degree of nonambiguity of a fuzzy set was defined in various classes of fuzzy measures and fuzzy partitions; entropy of a fuzzy partition was defined, and it was used to propose a new theory of information.

In the field of artificial intelligence, results were obtained in the following directions: pattern recognition and hierarchical classification; automated theorem proving and rewriting systems; symbolic computation; genetic algorithms.

Parallel computation and distributed system were other directions of research; the following themes were tackled: distributed data bases; concurent or parallel programming; distributed systems modelling.

In the field of operating systems, the problem of memory allocation was studied, a file system was proposed, and some Romanian books in this field have been written.

Relational data bases, and the retrieval of information from a data base were also studied. Algorithms for the approximation of curves and surfaces, and for computer graphics were given.

Also, some concrete problems in other fields of science were solved: classification, statistics, and simulation applied in agronomy, archaeology, chemistry, engineering, geography, geology, and celestial mechanics.

Some of our enthusiastic colleagues have implemented, in their free time, a lot of useful programs for the University, and many of us have solved problems from economy and industry. This is not a research activity but, at least in the first years, it had a serious impact on our teaching and research activity.

In the teaching activity there was an strong shortage of books for students. An important part of our time was used for printing our lectures as student courses or published books.

Certainly, everybody will continue to investigate his own field of research. There is the will to keep up with the world research in the directions mentioned above. We think there is a continuous need to improve our own programming paradigms, including object-oriented programming. There is a continuous need to contribute to the mathematical foundations of computer science. It is very important to investigate the data bases and public networks. In the future the declarative and nonprocedural programming can be more important than procedural programming. Also, we must keep open our mind to the problems of other fields of science.

References

[1] E. Dubinsky, *Mathematical Structures for Computer Science*, The American Mathematical Monthly, 91(1984), no.6, 379-381.

[2] G.E. Forsythe, *A University's Educational Program in Computer Science*, Comm.A.C.M., 10(1967), no.1, 3-6.

[3] M. Frenţiu, E. Munteanu, *Past and Future in Computer Science at the University of Cluj*, Preprint 1994.

[4] K. Harrow, *Theoretical and Applied Computer Science: Antagonism or Symbiosis*, The American Mathematical Monthly, 86(1979), no.4, 253-259.

[5] R.W. Hamming, *Intellectual Implications of the Computer Revolution*, The American Mathematical Monthly, 70(1963), no.4, 4-11.

[6] R.W. Hamming, *One Man's View of Computer Science*, Journal A.C.M., 16(1969), no.1, 3-12.

[7] J. Hartmanis, *On the Computational Complexity and the Nature of Computer Science*, Comm.A.C.M., 37(1994), no.10, 37-44.

[8] D.E. Knuth, *George Forsythe and the development of Computer Science*, Comm.A.C.M., 15(1972), 721-726.

[9] D.E. Knuth, *Computer Science and its relation to mathematics*, The American Mathematical Monthly, 81(1974), no.4, 323-343.

[10] D.E. Knuth, *Algorithmic Thinking and Mathematical Thinking*, The American Mathematical Monthly, 92(1985), no.3, 170-182.

[11] J.G. Laski, *Informatiks as a Science or the Science of Informatiks*, CC PAS Peports, 59, Warszawa, 1972.

[12] IBM J. of Research and Development, 25(1981), no.5, anniversary issue.

"BABEŞ-BOLYAI" UNIVERSITY, FACULTY OF MATHEMATICS AND COMPUTER SCIENCE, RO-3400 CLUJ-NAPOCA, ROMANIA

E-mail address: {muntean,mfrentiu}@cs.ubbcluj.ro

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

USING MAPLE IN PROCESSING POISSON EXPRESSIONS

B. PÂRV

Abstract. This paper presents an implementation of the Poisson Symbolic Processor by using the Computer Algebra System Maple and object-oriented facilities introduced by version 1.0 of Gauss package. An hierarchy of abstract classes (called *categories* in Gauss) is defined; each class represents an entity which is used to construct a Poisson expression (which is the partial sum of a Poisson series): monomial part, trigonometric part, term, expression. From each category there is derived at least one "concrete" class (called *domain* in Gauss). Each domain derived from the same category has its own representation (unknown at category level) and, possibly, new code for its operations (in order to achieve efficiency). This paper describes an experimental work; it constitutes the beginning of a long-term research. Computer algebra system Maple is used here as testing tool, due to its interactivity, user-friendly interface, and ready-to-use capabilities. After the experimentation work will be finished, we intend to develop an operational version of the processor using a high-level programming language.

1. Poisson series and expressions

A *Poisson series* have the form:

$$S = \sum_{i=0}^{\infty} C_i x_1^{j_1} x_2^{j_2} \cdots x_m^{j_m} \begin{matrix} sin \\ cos \end{matrix} (k_1 y_1 + k_2 y_2 + \cdots + k_n y_n), \tag{1}$$

where: C_i are *numerical coefficients*; $x_1, x_2, \cdots, x_m$ are *monomial variables*; $y_1, y_2, \cdots, y_n$ are *trigonometric variables*; $j_1, j_2, \cdots, j_m$ and $k_1, k_2, \cdots, k_n$ are exponents and coefficients, respectively. The summation index i covers the set of all possible combinations of exponents j and coefficients k ($j \in \mathrm{Z}^m$, $k \in \mathrm{Z}^n$).

One can write the form (1) of a Poisson series as follows:

$$S = \sum_{i=0}^{\infty} T_i, \tag{2}$$

Received by the editors: July 27, 1996.

1991 *Mathematics Subject Classification.* 68Q40, 68Q65.

1991 *CR Categories and Descriptors.* D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.2 [Programming Languages]: Language Classifications – object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – abstract data types; I.1.3 [Algebraic Manipulation]: Languages and Systems – special-purpose algebraic systems.

in which T_i is a *term* of this series:

$$T_i = C_i P_i F_i,$$

where the *polinomial part* P_i is:

$$P_i = x_1^{j_1} x_2^{j_2} \cdots x_m^{j_m}, \tag{3}$$

and the *trigonometric part* F_i have the form:

$$F_i = sin/cos(k_1 y_1 + k_2 y_2 + \cdots + k_n y_n), \tag{4}$$

In practice one does not operate with Poisson series, but with partial sums of these ones, the so-called *Poisson expressions*:

$$S = \sum_{i=0}^{N} T_i, \ N \in \mathrm{N} \tag{5}$$

This new version of the Poisson Symbolic Processor presented here is able to manipulate Poisson expressions of the form (5).

2. Maple and Object-Oriented Programming

The Maple programming language [2, 3, 4] is a procedure-based one. Despite this fact, due to the powerful data types available, one can simply implement all the basic concepts of the object-oriented programming. At least two Maple features facilitate this task:

1. interpretative nature of Maple, which means, from our viewpoint, dynamic binding of function calls, and
2. Maple table, in fact a data structure corresponding to the hash table, which is well-suited for implementing class definitions.

The Gauss package (see [5]), presented in the next section, constitutes an example of implementing object-oriented concepts in Maple.

Why object-oriented computer algebra? This question is not a new one, but possible answers were given only in the last decade [1, 5, 7]. The practical answers appeared a few years ago, especially IBM's AXIOM, and the Gauss package in Maple.

First, an object-oriented approach is well-suited for real-world modelling. In the case of computer algebra systems, the real objects are mathematical ones (numbers, functions, operators, functionals, sets, structures, more generally analytical expressions).

These objects have a well-defined behaviour (by axioms, rules, theorems, properties), and, finally, the goal of the mathematical science is to study this behaviour.

Second, the object-oriented approach uses classification mechanisms as tools for managing real world complexity. These mathematical objects can be grouped in classes, and the inheritance relations between classes can be expressed by the class hierarchies.

Third, an other mathematical-specific thing, the parameterization, was borrowed in the world of programming languages; the term used here is genericity. By using object-oriented programming and genericity, one can express, in a natural way, the class hierarchies of mathematical objects.

2.1. **Gauss Package.** The Gauss Package introduces a new approach of programming, based on parameterized (generic) data types.

There are two main concepts: category and domain. A *category* is an *abstract* and *parameterized* (*generic*) class, and a *domain* is a *concrete* class in the object-oriented terminology. An object is an instance of a domain. The category does not know the representation of objects - this is essentially the meaning of the *abstract* word; conversely, the domain (which represents a concretization of a category) defines the representation of its objects. The category definition contains: ancestor(s) specification, definition of signatures for all specific operations, and the implementation of all operations which can be implemented at category level (without knowing the representation). The domain definition defines domain category (categories), states the representation and implements the remaining operations (unimplemented at category level). Using multiple inheritance, Gauss implements a wide hierarchy of categories, all of them being algebraic objects. The root of this hierarchy is `Set` category, with the following operations:

1: `=, <>` – comparison operators

2: `Input` – conversion from Maple representation to domain representation (known as *initial constructor* in the object-oriented terminology)

3: `Output` – conversion from domain representation to Maple representation

4: `Random` – generates a pseudorandom value from domain

5: `Type` – predicate which defines the domain structure; it is used to test if an expression belongs to domain (from representation point of view).

2.2. **Abstract and parameterized classes.** Gauss domains are in fact parameterized (generic) classes. From implementation point of view, a domain is a Maple function, which

returns a Maple table with operation names as table entries. The domain ancestor(s), the class instances, and the properties of defined operations are also table entries.

For example, the domain `Integer()` returns the table in which the entries are: integer addition ('+'), subtraction ('-'), multiplication ('*'), greatest common divisor (gcd), zero and unit elements, etc.

The parameters of every domain are any valid Maple expressions, including other domains. The Gauss package contains a number of *usual* domains (see [5]). For example, the domain `DUP(R, x)` from Gauss (`DenseUnivariatePolynomial(R,x)` i.e. the domain of univariate polynomials with real coefficients) has two parameters: the coefficient ring (`R`) and the variable name (`x`). The first parameter, `R` must be a Maple domain, while the second must be a Maple name.

From programming viewpoint, a Maple category is the Maple (Gauss) implementation of an abstract data type. One can derive several domains from the same category, simply by choosing different representations (and then by implementing accordingly the corresponding operations). For example, Gauss package includes the `ExponentVector(X)`, `EV(X)` category, where `X` is a list of Maple names. This category defines the *exponent vector* abstract data type: if `X` contains the symbols $x_1, x_2, \ldots, x_n$, then `EV(X)` will define the set of all monomials in the variables `X`, i.e. monomials of the form:

$$x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n} \tag{6}$$

where x_j are variable names, and e_j are exponents (natural numbers). If the list `X` is ordered, and if this list is a parameter of the domain being considered, then for every monomial of the above form it is necessary to consider only the list (vector) of exponents $E = \{e_1, e_2, \cdots, e_n\}$.

One of the major benefits of the object-oriented programming is software reuse. Every (object-oriented) application framework contains a class hierarchy, from which, by using inheritance, one can extend (derive) new classes needed for an application-specific domain, with a minimum programming effort. In most situations, the existing classes are almost all what we need. By applying these considerations to the Gauss package (which is, in fact, an example of application framework), the programming steps are the following:

1. Define all the necessary categories

2. Define all the corresponding domains
3. Instantiate the domains by setting properly the generic parameters
4. Manipulate the obtained objects by using their operations.

2.3. **Deriving new categories and domains.** For a new category (for example `ExponentVector`), does not matter the representation of its objects; we are only interested in choosing the specific operations we need for manipulating the objects. The *natural* operations with monomials are multiplication, division, and common factor of two monomials. Consequently, besides the already inherited operations, the `ExponentVector` category must contain the definition of the following operations:

- addition of two exponent vectors (which corresponds to monomial multiplication)
- subtraction of two exponent vectors (which corresponds to monomial division)
- the minimum of two exponent vectors (which corresponds to monomial greatest common divisor).

The operations inherited from `Set` must be re-implemented. There are two alternatives: to implement general algorithms at category level (which, in turn, must invoke the representation-specific functions, defined at domain level), or *to postpone* (the used term is *to deferre*) this implementation to the domain level (where the representation is known).

After the category is completely defined, one can derive from it the corresponding domains. Usually, one can derive many domains from the same category. For example, in the `Gauss` package, there are presented four domains derived from the `ExponentVector` category: `DenseExponentVector`, `PrimeExponentVector`, `MapleExponentVector`, and `MacaulayExponentVector`. Every such a domain uses a different representation, and, consequently, specific algorithms for the operations it implements.

3. The class hierarchy for the Poisson Symbolic Processor

Based on the definition (1), the elements of a Poisson series (expression) can be defined in an hierarchical way. First, we consider the term components (factors): the coefficient, monomial part, and trigonometric part.

One can easy observe the elements well-suited for parameterization: the lists of monomial and trigonometric variables. For a concrete problem, these variables (symbols) are parameters of its corresponding mathematical model.

The coefficient is a rational number. Because Maple contains rational data type as a builtin data type, there are no problems in working with rational numbers; besides this, the `Gauss` package contains the definition of the `Rational` domain.

The monomial part can be considered as an instance of an exponent vector. Because the existing `ExponentVector` category deals with exponents considered natural numbers, one must redefine this category in order to manipulate integer exponents.

The trigonometric part has two components: the trigonometric function to be applied (it sin or *cos*) and its argument (a linear combination of trigonometric variables, with integer coefficients). For the same reason as in the case of `ExponentVector`, the argument can be expressed in the terms of a list of coefficients, if the list of trigonometric variables is considered as parameter. The `CoefficientVector` category described below implements the definition of this list (vector) of coefficients. Based on this category, one can define `TrigPart` category, which describes the behaviour of the trigonometric part.

Finally, the `PoissonExpression` category defines the behaviour of the Poisson expressions, taking the coefficient ring, the `ExponentVector`, and the `TrigPart` categories as parameters.

3.1. **The ExponentVector category.** This category is parameterized with the list of monomial variables (list of names). The operations are:

- `Variables` - returns the list of monomial variables
- `Index` - returns the index of the argument - a monomial variable name - in the list of monomial variables
- `Exponent` - returns the exponent of the argument - a monomial variable name -in the current object
- `Dim` - returns the number of monomial variables
- `Vect2List` - conversion from internal representation to list of integers
- `List2Vect` - conversion from list of integers to internal representation.
- *addition* - multiplication of monomial parts
- *subtraction* - division of monomial parts
- *comparison* - returns -1, 0 sau 1, i.e. less than, equal, or greater than.

The last two operations, `List2Vect` and `Vect2List`, are defined but not implemented in this category (by using object-oriented terminology, these function can be referred as *pure virtual functions*). By using these functions, one can implement, at category level, some of the defined operations (despite the fact one does not know the representation). For example, the algorithm for the addition of two exponent vectors consists of the following steps:

1. operand conversion from internal representation (unknown at this point) to list representation (natural representation of a vector) using `Vect2List`;
2. adding lists, element by element;
3. result conversion from list representation to internal representation using `List2Vect`.

Besides the above-discussed operations, the `ExponentVector` category definition also contains the implementation of some conversion functions, inherited from `Set`: `Input` and `Output`. Actually, these operations are implemented in all categories we discuss.

The following text represents the definition of `ExponentVector` category:

```
#
# ExponentVector(X)
#
ExponentVector := proc(X:list(name))
  local D, env;
  if not type(X, list(name)) then ERROR('invalid argument') fi:
  D := NewCategory();
  D := OrderedAbelianMonoid( op(D) );
  addCategory( D, ExponentVector );
  defOperation( Variables, List(Name), D );
  defOperation( Index, Name -> Integer, D );
  defOperation( Exponent, [D, Name] -> integer, D );
  defOperation( Dim, Integer, D );
  defOperation( List2Vect, List(Integer) &-> D, D );
  defOperation( Vect2List, D &-> List(Integer), D );
  defOperation( '+', [D, D] &-> D, D );
  defOperation( '-', [D, D] &-> D, D );
  defOperation('<>=', [D, D] &-> Union(-1,0,1), D );
# implement some operations...
  op(D)
end: # ExponentVector Category
```

For the reasons discussed above, there is another *deferred* operation, denoted by `'<>='`. Like `Vect2List` and `List2Vect`, this operation is used for implementing other comparison operations (`'<'` and `'='`).

3.2. **The CoefficientVector Category.** This category is parameterized with the list of trigonometric variables (list of names). The operations include:

- `Variables` - returns the list of trigonometric variables
- `Index` - returns the index of the argument - a trigonometric variable name - in the list of trigonometric variables
- `Coeff` -returns the coefficient of its argument - a trigonometric variable name - in the current object
- `Normalize` - normalizes the argument of a trigonometric function
- `Dim` - returns the number of trigonometric variables
- `Vect2List` - conversion from internal representation to list of integers
- `List2Vect` - conversion from list of integers to internal representation
- *addition* - addition of trigonometric function arguments
- *subtraction* - subtraction of trigonometric function arguments)
- *multiplication* with a rational number
- *comparison* of two trigonometric coefficients.

Because the linear combinations of trigonometric variables are arguments of the trigonometric functions *sin* and *cos*, these combinations need to be *normalized.* Taking into account the parity of these functions, we define the *normal form* of an argument the form in which the first non-zero coefficient is positive. The `Normalize` operation returns the normal form of its argument, and a state value (-1 or 1) which indicates the sign of the first non-zero coefficient in the initial argument.

The following text represents the definition of `CoefficientVector` category:

```
#
# CoefficientVector(X)
#
CoefficientVector := proc(X:list(name))
local D, env;
if not type(X, list(name)) then ERROR('invalid argument') fi:
D := NewCategory();
D := OrderedAbelianMonoid( op(D) );
addCategory( D, CoefficientVector );
defOperation( Variables, List(Name), D );
defOperation( Index, Name &-> integer, D );
defOperation( Coeff, [D, Name] &-> rational, D );
defOperation( Normalize, [D, integer] &-> D, D );
defOperation( Dim, Integer, D );
defOperation( List2Vect, List(rational) &-> D, D );
defOperation( Vect2List, D &-> List(rational), D );
defOperations( {'+', '-'}, [D, D] &-> D, D );
defOperation( '.', [rational, D] &-> D, D );
```

```
defOperation( '<>=', [D, D] &-> Union(-1,0,1), D );
# implement some operations...
op(D)
end: # CoefficientVector Category
```

3.3. **The TrigPart category.** The `TrigPart` category has as argument the domain of coefficient vectors, which in turn must be derived from `CoefficientVector` category. The operations are:

- `List2Trig` - conversion from list to internal format
- `Trig2List` - conversion from internal format to list
- `CoefficientVector` - returns the CoefficientVector domain
- `Variables` - returns the list of trigonometric variables
- `Argument` - returns the coefficient vector of an object
- `Function` - returns the trigonometric function
- `Coeff` - returns the coefficient of its argument - a trigonometric variable name - in the current object
- `Dim` - returns the number of trigonometric variables
- `sin, cos` - apply the trigonometric functions on an `CoefficientVector` object
- *multiplication* of two trigonometric parts
- *comparison* of two trigonometric parts.

In the case of multiplication, in order to preserve the form of the trigonometric part, one must apply the following rules:

$$sin(x) * sin(y) = (cos(x-y) - cos(x+y))/2,$$
$$sin(x) * cos(y) = (sin(x-y) + sin(x+y))/2,$$
$$cos(x) * cos(y) = (cos(x-y) + cos(x+y))/2.$$

The following text represents the definition of `TrigPart` category:

```
#
# TrigPart
#
TrigPart := proc()
  local S, P, T, env;
  S := args[1];
  if not hasCategory(S, CoefficientVector)
    then ERROR('the argument must be an CoefficientVector') fi;
  P := newCategory();
  addCategory(P, TrigPart);
  defOperation( List2Trig, Record(Name, S) &-> P, P);
  defOperation( Trig2List, P &-> Record(Name, S), P);
  defOperation( CoefficientVector, CoefficientVector, P);
  defOperation( Variables, List(Name), P);
```

```
  defOperation( Argument, P &-> S, P);
  defOperation( Function, P &-> Union(cos, none, sin), P);
  defOperation( Coeff, P &-> Rational, P );
  defOperation( Dim, Integer, P);
  defOperations( {sin, cos}, [Rational, S] &-> [Rational, P], P );
  defOperation('*', [P, P] &-> List(List(Rational, P)), P);
  defOperation('<>=', [P, P] &-> Union(-1, 0, 1), P);
# implement some operations...
  op(P)
end: # TrigPart Category
```

3.4. **The PoissonExpression Category.** The `PoissonExpression` category has the following parameters:

- the coefficient ring (usually the ring of rational numbers)
- the exponent vector domain
- the trigonometric part domain.

and the defined operations are:

- `List2Expr` - conversion from list to internal format
- `Expr2List` - conversion from internal format to list
- `CoefficientRing` - returns the coefficient ring domain
- `ExponentVector` - returns the exponent vector domain
- `TrigPart` - returns the trigonometric part domain
- `PolVariables` - returns the list of monomial variables
- `TrigVariables` - returns the list of trigonometric variables
- `PolDim` - returns the number of monomial variables
- `TrigDim` - returns the number of trigonometric variables
- *multiplication* of a term with a rational number
- *addition* (n-ary operation)
- *multiplication* (n-ary operation)
- `comparison` of two terms
- `Diff` - partial derivative with respect to a specified variable
- `Int` - integration with respect to a specified variable.

The following text represents the definition of `PoissonExpression` category:

```
#
# PoissonExpression
#
PoissonExpression := proc()
  local R, E, S, P, T, env;
  R := args[1];
```

```
  E := args[2];
  S := args[3];
  if not hasCategory(R, Ring)
  then ERROR('1st argument must be a Ring')
  elif not hasCategory(E, ExponentVector)
  then ERROR('2nd argument must be an ExponentVector')
  elif not hasCategory(S, TrigPart)
  then ERROR('3nd argument must be an CoefficientVector') fi;
  P := newCategory();
  addCategory(P, PoissonExpression);
  if hasCategory(R, UFD) then P := UFD(P)
  elif hasCategory(R, GcdDomain) then P := GcdDomain(P)
  elif hasCategory(R, IntegralDomain) then P := IntegralDomain(P)
  elif hasCategory(R, CommutativeRing) then P := CommutativeRing(P)
  else P := Ring(P)
  fi;
  defOperation( List2Expr, List(Record(R, E, S)) &-> P, P);
  defOperation( Expr2List, P &-> List(Record(R, E, S)), P);
  defOperation( CoefficientRing, Ring, P);
  defOperation( ExponentVector, ExponentVector, P);
  defOperation( TrigPart, TrigPart, P);
  defOperation( PolVariables, List(Name), P);
  defOperation( TrigVariables, List(Name), P);
  defOperation( PolDim, Integer, P);
  defOperation( TrigDim, Integer, P);
  defOperation( '.', [R, P] &-> P, P );
  defOperation( '+', Nary(P) &-> Nary(P), P);
  defOperation( '*', Nary(P) &-> Nary(P), P);
  defOperation('<>=', [P, P] &-> Union(-1, 0, 1), P);
  defOperations( {Diff, Int}, [P, Name] &-> P, P);
# implement some operations...
  op(P)
end: # PoissonExpression Category
```

4. Implementation issues

Two different domains were implemented for each of the above-discussed categories. Each implementation uses a different internal representation of the exponent vector and coefficient vector: *list* and *Gödel coding.*

In the case of *list representation* every exponent or coefficient vector is represented as a Maple list, in which all the elements are integer numbers (Maple integers). Consequently, the `List2Vect` and `Vect2List` operations will implement the identity function and the comparison operator is based on the lexicographic ordering.

In the case of *Gödel coding*, an exponent or coefficient vector is represented as rational number, by using the first n prime numbers (n is the number of elements in the vector):

$$(a, b, c, d, \ldots) \rightarrow 2^a 3^b 5^c 7^d \cdots$$

Taking into account the integer nature of vector elements (positive or negative numbers), it follows that the result of this mapping is a rational number; its numerator will contain the product of the powers with positive exponents, while the denominator will include all the powers with negative exponents. The function which define this coding is bijective, so it follows that this representation is unique and it exists the inverse transformation. Thus, one can define the `List2Vect` and `Vect2List` operations as follows: `List2Vect` will implement the direct mapping, while `Vect2List` the inverse one. The considered ordering is the natural ordering of the rational numbers.

Domain implementation also contains the `Type` operation (inherited from `Set`; it returns a boolean value, depending on whether its arguments satisfies the rules stated for the concrete representation chosen), and its zero element `D[0]`.

Finally, the aritmetic operations were redesigned, in order to exploit the specific features of the concrete representation. After all, the invoking of the conversion routines `List2Vect` and `Vect2List` was removed, in order to gain computing speed.

The implemented domains are: `ListExponentVector` and `PrimeExponentVector`, `ListCoefficientVector` and `PrimeCoefficientVector`, `ListTrigPart` and `PrimeTrigPart`, `ListPoissonExpression` and `PrimePoissonExpression`.

References

[1] S.K. Abdali, G.W. Cherry, N. Soiffer, *An Object Oriented Approach to Algebra System Design*, ISSAC-86, 24-30, 1986.

[2] B.W. Char et al., *Maple V – Library Reference Manual*, Springer, 1992.

[3] B.W. Char et al., *Maple – Language Reference Manual*, Springer, 1992.

[4] B.W. Char et al., *First Leaves: A Tutorial Introduction to Maple*, Springer, 1992.

[5] D. Gruntz, and M. Monagan, *Introduction to Gauss*, MapleTech 9, 1993, 23-35.

[6] R. Jenks, and R. Sutor, *AXIOM – The Scientific Computation System*, Springer, 1992.

[7] M. Monagan, *Signatures + Abstract Data Types = Computer Algebra – intermediate expression swell*, PhD Thesis, University of Waterloo, 1989.

[8] B. Pârv, *Poisson Symbolic Processor*, Studia, Mathematica, XXXIV, 1989, No. 3, 17-29.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: bparv@cs.ubbcluj.ro

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

AN ACTIVE OBJECT MODEL FOR MULTIMEDIA PRESENTATIONS

M. PAPATHOMAS AND V.-M. SCUTURICI

Abstract. In this paper we propose an active object model that aims at a more comprehensive approach in integrating concurrent programming and object-oriented features. The model incorporates a number of previously proposed features with the novel features of abstract states, state predicates and state notification. We have started using the prototype for the development of multimedia programming environment based on active objects. A prototype of the model has been implemented in Python and the presentation in this paper is based on this implementation.

1. Introduction

Substantial research activity in the past few years concentrated on the design of languages and models for integrating concurrency and object-oriented features with the intention to enhance the potential for software reuse in the development of concurrent systems. Most of the work in the area has focused on the problem of combining inheritance with concurrency and more particularly specifying and reusing through inheritance synchronization constraints on the invocation of objects' methods. Currently, this is a widely recognised problem and several solutions have been proposed. However, most proposed solutions address this problem in isolation. More recent and less research has addressed the issue of coordinating the execution of a set of objects and of specifying and reusing coordination patterns separately from objects. Furthermore, few languages supporting the proposed features are widely available and relatively little experience has been gained from their use.

In this paper we propose an active object model that aims at a more comprehensive approach in integrating concurrent programming and object-oriented features. The model incorporates a number of previously proposed features with the novel features of *abstract*

Received by the editors: September 12, 1996.

1991 *Mathematics Subject Classification.* 68Q10, 68Q90.

1991 *CR Categories and Descriptors.* D.1.5 [Programming Techniques]: Object-oriented Programming; D.1.3 [Programming Techniques]: Concurrent Programming; D.2.6 [Software Engeneering]: Programming Environments; D.3.2 [Programming Languages]: Language Classifications – object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structures; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems.

states, *state predicates* and *state notification*. The model has been designed for addressing simultaneously the following reuse aspects in concurrent object-oriented programming:

- Support for self-contained objects that can be reused across applications.
- Reuse of methods through class inheritance.
- Reuse of synchronization constraints.
- Composition of active object behaviours.
- Object coordination.

A prototype of the model has been implemented in Python and the presentation in this paper is based on this implementation. Python is freely available on a large number of systems. This will allows us to make our prototype widely available and gain more experience with the use of its concurrent object-oriented features in the development of applications.

2. The Object Model

Objects are active entities that resemble server processes that accept requests from other object, they can delay requests and process them in an order that is most suitable to them. Requests are processed by threads that execute quasi-concurrently within an object. Threads may also be created spontaneously at the creation of an object.

The main aim in the design of this object model is to enhance the potential for reuse in concurrent object-oriented systems by integrating support for all the reuse issues discussed in section 2. For doing so, the model incorporates a number of message passing features that are combined with thread scheduling in such a way that allows objects to schedule the processing of requests and replies in a self-contained way. These features are integrated with the novel concepts of abstract states, state predicates and state notification. The use of these features avoids the problems caused by the use of inheritance in COOPLs, supports new ways to specify and inherit active object behaviours and also provides support for object coordination.

The model is general enough so that it may be incorporated in a variety of languages. The current version is implemented as an extension to the language Python. The linguistic constructs and examples used in this paper are based on the Python implementation.

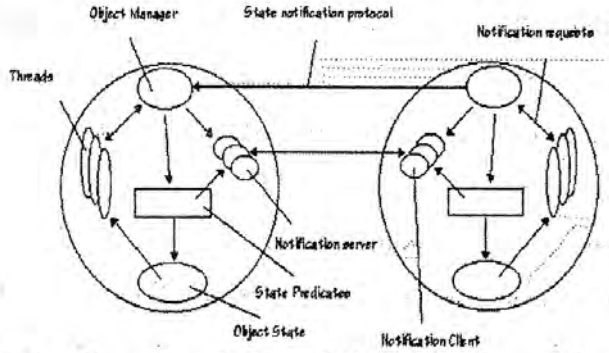

FIGURE 1. Conceptual view of the object execution model

2.1. **Language-Independent Description.** Figure 1 shows a conceptual view of object execution in our model. Each object is associated with an object manager that controls the actions that are executed by its object. The object manager is a conceptual entity that makes it easier to describe the behaviour of objects in our model.

The object manager instructs its object to carry out a number of actions such as to start or resume a thread that executes some methods of the object. The object executes the code of a method until the occurrence of an event, such as the completion or suspension of a method. The occurrence of such an event activates the object manager which decides what should be the next action to be executed by its object. The object can thus be either executing some thread or it may be waiting for the object manager to instruct it what it should to do next. In the latter case, we will say that the object is at a stable state.

The execution of an object can be seen as a graph where nodes represent stable states and edges are associated with tuples of the form (a,e) where a is the action that the object was instructed to execute in the previous stable state and e is the event that stopped the execution of a and made the object to move into the next stable state.

Apart from events that are generated from the execution of its object, the object manager may also be activated by external events that are generated by other object in program. The events that trigger the execution of the object manager are:

- *A method invocation request is received at the object.* In this case the object manager creates a new thread for running the method. If there is no thread already active within the object and if the object is at a state where it can execute the request method, the object manager runs the newly created thread.

- *A thread completes the execution of its method.* In this case the object manager chooses another waiting thread, if any, for execution.
- *A thread requests to be suspended until the occurrence of an event.* In this case the object manager removes the thread from the queue of threads that are ready to run and inserts it in a queue of threads that are waiting for an event. Then, it picks another thread, if any, for execution. There are several types of events a thread may wait for. For instance, it may wait for the reply to a request it has issued to another object or it may wait until the object reaches a certain state. These events will be discussed later in conjunction with the constructs generate them.
- *The occurrence of an external event of interest to one of its threads.* This any event awaited by some of the object's threads that is not generated by actions that are executed within the object. This can be for instance: the arrival of the reply to a request a thread has made to another object or the arrival of a state notification event (to be discussed later). In this case, the object manager removes the thread from the list of waiting threads and depending on whether or not the thread can be run immediately it resumes the execution of the thread.
- *The receipt of a state notification request.* This is a request by another object manager asking the object manager to generate a notification event when its object reaches a certain state. The object manager stores the request in a queue of notifications requests and when its object reaches the request state sends a notification event. State notification is a novel feature of our model that will be discussed in detail later.

In many of the situation discussed above, the object manager needs information about the object's state. In our model, however, the object manager does not see or access the object state directly. The concept of *abstract state* is used to represent properties of interest with respect to the object's state at a level of abstraction that hides implementation details. *State predicates* provide the mapping from abstract states to the concrete object state. In order to find out whether the object is at an abstract state, the object manager goes through a state predicate. Abstract states and state predicates are discussed in detail next. Figure 1 shows a conceptual view of object execution in our model illustrating the interactions of the object components of model.

2.1.1. *Abstract States and State Predicates.* An abstract state represents some aspect of the real state of execution of an object at a level of abstraction that hides implementation details. The state of execution is taken here in a broad sense. It may comprise not only the values of the object's instance variables but also the messages that are suspended at the object interface, the state of execution of the object's threads, etc.

At each stable state in the object's execution graph, the object's manager decides what action will be executed next based on abstract states. In our model abstract states are used to constrains the execution of the possible actions. This is accomplished by allowing the programmer to constrain the acceptance of methods and to express the suspension or resumption of threads using abstract states. The linguistic constructs that are provided for doing so are discussed in the remaining of the document.

State predicates provide an interpretation for abstract states. They are used to tell whether or not a property that is represented by an abstract state holds a concrete state. The mapping from abstract states to concrete states provided by state predicates may differ for different classes and thus supports polymorphism for abstract states.

A state predicate can be defined as a function P from CS X Asp to {*true, false*}, where CS is the set of concrete object states and Asp is a subset of the set of the object's abstract states AS. The following are some important properties of state predicates:

- A set of abstract states may be true at one object state.
- An object may be associated with a set of state predicates and each predicate is associated to a subset of the object's abstract states.
- The subsets of abstract states associated with different state predicates of an object are disjoint.

From a practical point of view, state predicates are objects that encapsulate the information necessary to determine whether or not the property described by the abstract state is true at a certain stable state in the execution of the object. They are defined by the programmer and used by the object manager to evaluate the conditions that constraint the execution of the object's actions.

2.1.2. *State Notification.* State notification allows active objects to monitor and synchronize with state changes, expressed in terms of abstract states, occurring in other objects. State notification is a protocol provided by object managers that allows a thread in one

object to synchronize it's execution with state changes expressed in terms of abstract states of another object.

Figure 1 shows a conceptual view of the architecture used to support state notification. The figure shows two active objects A and B. B's object manager has made a state notification request to A's object manager. Following this, A's object manager has created a local notification server object that represents the notification request. The object manager maintains a list of notification server objects (notification requests); each time the object state changes it goes through the list and activates each notification server. The notification server evaluates its associated abstract state expression by invoking the appropriate state predicate(s) and informs the notification client if appropriate. When the notification client is informed, it requests its object manager to take the appropriate action; for instance, schedule a suspended method for execution.

State notification may be *asynchronous* or *synchronous*. In the asynchronous case, when the object is at stable state, its object manager services all notification requests and then proceeds with the normal execution of the object's methods. The synchronous variant allows the object that requested the notification and the notifying objects to be synchronized in a way similar to "rendez-vous". This variant guarantees, by postponing the execution of the notifying object's methods, that the notified object will get the chance to invoke methods of the notifying object while it is still at the requested state.

Together with abstract states, state notification may be used to describe abstractly, as a sequence of abstract state changes, practically any activity encapsulated within an active object. Also, note that this is a general architecture and can thus be instantiated with different implementations of notification clients and servers. For example, support for real-time notification can be provided by specifying a time-out at the notification client and/or by requesting a bound on the notification delay.

2.2. **Linguistic Support.**

2.2.1. *Abstract States.* In the implementation of our object model in Python, abstract states are represented as tuples that contain at least one element. This first mandatory element is a string that represents the name of an abstract state. For example, ('empty',) and ('full',) are used to represent the abstract states of the bounded buffer. Apart from the first mandatory string element that represents the state name, a tuple that represents an abstract state may contain an arbitrary number of additional elements. In this case

the abstract state name may be used to represented a family of abstract states that are further qualified by the additional elements. The semantics of these additional elements is defined by the state predicate responsible for the abstract state. For instance, a set of abstract states ('contains',n), where n is a natural number, may be used to represent the states where a container object contains exactly n elements. In the buffer example it is the functions empty and full that are implicitly defined as state predicates for the states with the same name. Later we will see other ways for defining state predicates explicitly.

2.2.2. *State Predicates.* State predicates are objects that are associated with an object and are responsible for determining whether or not the associated object is at a given abstract state. An object may have a set of state predicates and each state predicate is associated with a disjoint subset of the set of abstract states defined for the object. When the object manager needs to determine if an object is at an abstract state, it calls the state predicate that is responsible for that particular abstract state.

If no state predicate is specified explicitly for an abstract state the object should have a method with the same name as the state that is used to determine whether or not the property associated with is abstract state is true.

State predicates may be associated with an object statically or dynamically. It is also possible for a state predicate to be shared among several objects. This feature can be used as it will be shown in section 4.4.1 to support object coordination.

In a class definition, the variable state_predicates is used to associate abstract states to state predicates. This variable should be set to a dictionary, entries of this dictionary should have as key a state predicate class and as value a list abstract states. For abstract states that are not associated explicitly with a state predicate in the state_predicates variable, a method with the same name as the abstract state should be provided for allowing to determine whether or not the object is at the associated state.

The method newPred supported by all active objects may be used to associate a state predicate to an object instance at run-time. This method takes as arguments a state predicate object and a list of the associated abstract states. The object should not define these any of these states already.

In order to be able to operate with the object manager, state predicates should define a method evalState that takes as argument a tuple that represents an abstract state and the object for which it is needed to find out if it is at the specified state. This

method should return None to indicate that the object is not at the requested state or a value different to None otherwise.

In order to determine whether or not its associated object is at a requested abstract state, the state predicate has to get some information about its object. It's part of the definition of the state predicate to state clearly the nature of this information and it's the responsibility of the programmer that defines the active object class that uses a particular state predicate to make sure that its class makes available the information required by the state predicate to determine whether or not the object is at a given state.

2.2.3. *Activation Conditions.* Activation conditions are used to constrain method invocations and more generally the execution of the object's threads. They associate methods with a condition, expressed in terms of abstract states, that has to be true in order to run a thread that executes the associated method. Activation conditions may be associated to an object either statically, in it's class definition, or dynamically to a particular instance at run-time.

In the definition of a class, the variable conditions is used to specify activation conditions. This variable should be set to a dictionary having as keys objects that designate a set of methods the acceptance of which is constrained by the condition, and as values, boolean functions that specify the conditions. The following types of objects may be used as keys:

- A string: in this case the string is the name of the method that is constrained by the condition.
- A list of strings: the strings have to be method names and in this case all methods in the list are constrained by the condition.
- A function: the function has to evaluate to a list of strings designating object methods. Such functions are evaluated at the creation of the active object class to determine the actual set of methods that is constrained by the condition. The function may use in its evaluation variables defined by the object that contain lists of method names as well as predefined functions that return such lists. For instance, the function allMethods may be used to return the list of all methods of the object.

Conditions are boolean functions that may use the predefined function atState that takes as argument an abstract state specification and returns the values true or false

depending on whether or not the object is at a state that matches it's argument. Abstract state specifications and the matching depend on the state predicate that is associated with a particular state.

Activation conditions may be associated with an instance at run-time by calling the method addCondition. This method is provided by the object manager and is supported by all active objects.

2.2.4. *Synchronization Actions.* The programmer can define a number of actions to be executed when certain events, such as the receipt of a request or the completion of a method execution, take place during the execution of the object. In these actions the programmer may use variables especially defined for this purpose to keep track of the occurrence of such events. These variables may then be used in the definition state predicates. This mechanism in combination with abstract states and state predicates, may be used to specify synchronization based on *synchronization counters.*

Some of these actions could be included directly in the code of object. However, this approach has several disadvantages. Most importantly, one can not anticipate all the information that will be needed by state predicates to be defined in subclasses when writing the methods of a class the first time. If more information is needed for defining the state predicates in a subclass, it will be necessary to redefine the methods inherited from the class. In addition, the code of the object's methods would become more complex as it would mix computations that have to carried out by the method as well as computations that are used to keep track of such events.

Such a feature has been presented in previous proposals [25][15]. In these, pre-actions and post-actions can be associated with methods and are executed before and after the execution of their associated methods. A more detailed discussion of the benefits of a such feature may be found in these references. Our proposal extends these previous proposals by the inclusion of further actions and provides more flexibility for the specification and inheritance of actions.

In addition to pre-actions and post-actions we introduce actions that are executed when a method invocation request is received by an object, actions that are executed when the execution of a method is suspended and resumed.

Synchronization actions are associated with methods by the definition of the dictionaries pre_action, post_actions, suspend_actions, resume_actions and receipt_actions.

The keys in the dictionary are either lists of method names or functions that evaluate to lists of method names. The values in the dictionary specify the actions associated with these various events in the execution of methods.

2.2.5. *Thread Creation and Synchronization. Thread Creation.* In addition to threads that are created to execute method invocation requests, it is possible for objects to have a number of threads that are created when an object is instantiated. These threads may execute quasi-concurrently with each other and with the threads that execute method invocation requests. The execution of these threads is constrained in the same way as the threads that execute requests.

The variable **activities** is used in the definition of a class to specify the list of methods that are to be executed by threads created spontaneously at the creation of an instance of the class. If the object's class defines a method with the name **Activity**, this is also executed in new thread at the creation of the object. The execution of such threads in constrained by activation conditions in the same way that threads that execute method invocation requests are. In the definition of a class it is possible to specify whether or not the activities of parent classes are inherited.

Synchronizing threads with abstract states of the object. A thread may suspend its execution until the object reaches a specified state by calling the method **suspendUntill** providing as argument an abstract state expression that specifies the requested state. This feature is particularly interesting when combined with the execution of background threads. Such a thread may loop waiting for the object to reach some state where it executes some background actions. The use of these features for defining and reusing active object behaviours is shown in the examples in section 4.

Synchronization with requests sent to other objects. The method **sendAndSuspend** may be used to issue a request to another object and suspend the calling thread until a reply is returned by the called object. This allows the calling object to do some other actions in a background thread or possibly accept other request while the called object processes its request. This feature provides some flexibility for reply scheduling and addresses the concurrent programming problems known as remote delays and nested monitor calls.

Asynchronous State Notification. An object that wants to be notified when another object reaches a state that satisfies an abstract state expression has to first issue a

notification request to the source object. The request returns an object that represents the notification event. This object may then be used to suspend a thread until the event occurs.

A notification request to an active object is made by calling the method **notifyRequest()** or the method **atState()**. This method takes as argument an abstract state expression and returns an object that represents the notification event. The method **suspendUntil()** and **waitUntil()** take as argument an object representing a notification event and suspend the calling thread until the event occurs. With the former, while the calling thread is suspended other threads in the object may be scheduled for execution. The execution of the suspended thread is resumed sometime after the event has occured and when there are no other active threads within the object. With the latter, no other threads may be scheduled while the thread waits for the event. In this case the thread is resumed immediately after the event occurrence.

It is also possible to wait for a combination of notification events. This is accomplished by the method **suspendUntilComplexEvent**. This method takes two arguments. The first argument is a dictionary that has keys strings used to tag the events and as values notification event object returned from calls to **notifyRequest**. The second argument may be one of the strings 'Any' or 'All'. In the first case, the complex event occurs when any of the notification events occurs. In the second case, the thread waits until all events have occured. The method returns the tag of the last event. This is useful in the case of 'Any' to determine which among a set of mutually exclusive events has occured.

Synchronous Notification. The syncBlock method allows the execution of a block of code in synch with one or more objects at a given state. For instance, the code below specifies that the code following 'do': will be executed when the object, anObject, is at the state: aState. The local variable theObject is bound to the object when the block of code following 'do': is executed.

```
self.SyncBlock(
  {'with': {'theObject': (anObject, ('aState',)),},
   'do': ''theObject.aMethod()''
   })
```

Once the object, anObject has reached the request state it will not accept any more calls at its interface. Method calls are only accepted through a privileged interface that is only known in the syncBlock code and is bound to a variable local to the block

- the variable theObject in the example. After the code block is executed the privileged interface is discarded and messages are again accepted at the ordinary object interface.

Another feature of the privileged interface used in the syncBlock is that activation conditions for methods are interpreted as assertions. That is, it is an error if the activation condition for a method is not satisfied whwn a method is called through this interface.

The reason for interpreting activation conditions as assertions is that it does not make sense to suspend a method call within a syncBlock statement. As the object will not accept any other calls on its ordinary interface its state may not change but only through calls executed within the syncBlock.

2.2.6. *Message Parsing Features. Issuing Requests.* Active object's methods may be invoked using the ordinary Python method invocation syntax which has in our model the semantics of a remote procedure call. However, the model provides other constructs allowing more flexibility in structuring object interactions. The message passing features provided are the following:

- Remote procedure call: this is supported by the ordinary method call syntax of Python. The calling object is blocked until the calling object replies
- Asynchronous method call: this is supported by the method send which, takes as argument a dictionary that represents the message to be sent. The field of this dictionary are explained below.
- Non-blocking remote procedure call: this is supported by the method sendAndSuspend that takes as argument a dictionary representing a message. The difference with remote procedure call is that while the thread that issues the request is suspended other threads may run in the object. The suspended thread is resumed after the reply has been received, there is no other active thread in the object and the object ia at a state where the method associated with thread can be run. The non-blocking designation assigned to this feature should be understood with respect to the object. The calling thread itself is blocked.

The dictionary that is used to represent the message to be sent by the Send and sendAndSuspend method contains the following fields:

- target: the object to which the message is sent
- key: a string specifyng the name of the method to be called
- args: a tuple with the arguments to be passed to the method

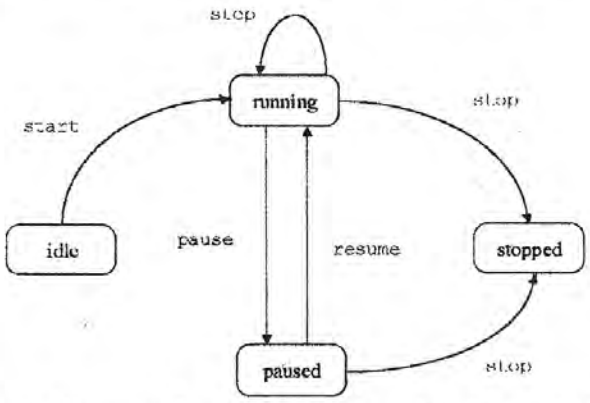

FIGURE 2. State diagram for the behavior of objects of class activity

- replyTo: this field is optional. Its value may be set to None. In either case, if it is present, the message is send asynchronously. In the case where it's set to a reply destination object other than None, the reply to the request is delivered to the caller associated with this reply destination.

3. Examples

3.1. **Combining Behavioural Patterns through inheritance.** Previous proposal for combining concurrency features with class inheritance, considered that, in order to be able to reuse methods in subclasses through inheritance, methods should not contain any synchronisation code. With the features provided in our model it is possible to use inheritance for reusing methods that contain synchronisation code. In this section we show, not only, that is not a problem but that in addition, this possibility also enhances reuse by allowing the definition and reuse of mixins that define behavioral patterns.

We define a set of abstract classes and mixins that specify and allow the reuse of the behavior of objects that representing continous activities.

A Basic Activity

The most general such behaviour is defined by the class Activity. The behavior of instances of this class is shown abstractly in the state diagram in Figure 3. Objects of class Activity may be at the states idle, running, paused and stopped. When first created, they are at the state idle where they can accept the method start and move at the state running. While they are in the state running they continously execute their method stepaction. This method should be redefined by subclasses and it corresponds to the actions to be executed by the activity at each step. At the state running objects accept calls to their pause and stop methods. The execution of the pause method moves

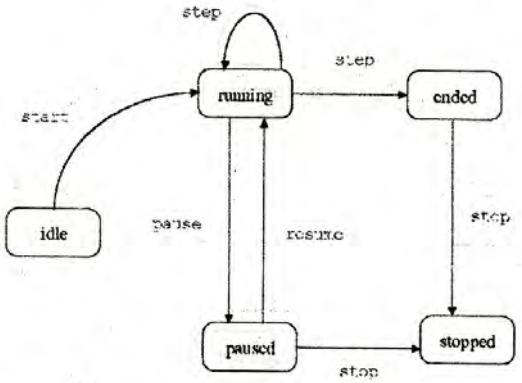

FIGURE 3. Behavior of an activity that ends

the object to the state paused where the execution of their activity is ceased temporarily. From the paused state an object may accept a resume method call and move back to the running state where it resumes the execution its activity. At the states running and paused, an object may accept a call to stop which moves the object into the stopped state. After moving to this state the object will not accept any further requests. Moving to this state triggers the execution of internal actions for freeing the resources used by the object.

Activities that Terminate after a Number of Steps. The class ActivityWithEnd refines the behavior of basic activities to specify the behavior of activities that terminate after a finite number of steps. The behavior of such an activity is illustrated by the state diagram in Figure 2. The state diagram includes a new state ended where an instance of ActivityWithEnd may move into after executing the last step of activity. Note that the state ended is different than the stopped state. Consider for instance an object encapsulating the playback of a video clip. The activity ends after the last frame of the clip fas been displayed by stepaction. However, the window showing the last frame may remain visible on the display. However, if the object moves into the stopped state, the window showing the last frame disapears.

The definition of the class ActivityWithEnd is shown in Figure 4. This class can be inherited by a class that also inherits Activity to obtain the behavior of an activity that ends after a finitenumber of steps.

This class defines two instance variables length and step. Length stores the number of steps of the activity and step stores the number of steps that were executed so far by the activity. The post-actions associated with stepaction is used by the state predicate to determined if the object is at the state ended. This function compares the number of

```
class SimpleActivity(ActiveObjectSupport):
  methods = ['pause', 'start', 'stop', 'resume', 'stepaction']
  states = ['idle','running', 'paused', 'stopped']
  conditions = {'start' : (lambda o: o.atState(('idle',)),),
                'stepaction' : (lambda o: o.atState(('running',)),)}
   state_predicates = {enum_states: [ 'idle','running', 'paused', 'stopped']}
   enum_states_afuncs = {'idle': 'isIdle', 'running' : 'isRunning',
                         'paused' : 'isPaused','stopped' : 'isStopped'}
   method_call_opt = ['stepaction']
   def __init__(self, stepDelay = 2):
     self.step = 0, self.running = 0, self.idle = 1, self.paused = 0
     self.active = 0, self.stopped = 0, self.steptime = stepDelay
  def isIdle(self,state): return self.idle
  def isRunning(self,state): return self.running
  def isPaused(self,state): return self.paused
  def isStopped(self,state): return self.stopped
  def Activity(self):
    #self.run = self.interface.notifyRequest(('running',))
    while not self.isStopped(1):
      self.step = self.step + 1
      self.sendAndSuspend({ 'target' : self.interface,
                            'key' : 'stepaction', 'args' : ()})
  def stepaction(self):
    print "executing step: %d ... \n" % self.step
    time.sleep(self.steptime)
  def pause(self): self.running = 0, self.paused = 1
  def start(self): self.active = 1, self.running = 1
  def resume(self): self.running = 1, self.paused = 0
  def stop(): self.running = 0, self.active = 0
class ActivityWithEnd(SimpleActivity):
  length = 10, states = ['ended']
  state_predicates = { enum_states : [ 'ended']}
  enum_states_afuncs = {'ended': 'isEnded'}
  conditions = {'stepaction': (lambda o: not o.atState(('ended',)), 'and')}
  def isEnded(self,state): return self.step >= self.length
class loopingActivity(ActivityWithEnd):
  loop = 0, states = ['looping']
  state_predicates = { enum_states : [ 'looping']}
  enum_states_afuncs = {'looping': 'inloop'}
  conditions = {'stepaction': (lambda o: not o.atState(('ended',))
              or o.atState(('looping',)),'and')}
  methods = ['looptoggle']
  def isEnded(self,state): return self.step >= self.length
  def inloop(self,state): return self.loop
  def looptoggle(self): self.loop = not self.loop
```

FIGURE 4. Continous activity with an end. Definition of a continous activity. Looping activity.

steps executed by the activity to the number of steps of the activity and returns true if they are equal.

This class defines a new activation condition that is combined with inherited activation conditions to constrain the execution of the method step action. This condition

ensures that the activity will not execute any more steps after it has reached its end - has executed its total number of its steps.

Activities that loop. The class loopingActivity can be combined with the previous classes to specify the behavior of activities that can loop. This class introduces a new abstract state looping and two new methods looptoggle and reset. The method looptoggle may be accepted at any abstract state of the object other than stopped and moves the objects from the abstract state looping to the abstract state not looping. This method does not affect the other abstract states of the object. If during its execution an object moves to a state corresponding to the abstract state ended and looping, the reset method is executed and the object moves to the state running where it starts executing its activity from the beginning. The reset method has to be defined in subclasses and it should include the actions needed for restarting the activity. For instance, if the object plays a video clip that is stored in a file, the reset action moves the file pointer to the beginning so that its stepaction method will redisplay the video frames from the beginning of the video clip.

Figure 4 shows the definition of the class loopingActivity. This class defines the function looping that is used to move into and out of this state. The method ResetAndEnd is executed by a thread created spontaneously at the creation of the object - this is specified by the definition of the activity variable. This thread waits until the object at a state where the abstract states looping and ended are true simultaneously. Then, it calls reset to execute the actions that will allow the activity to restart executing from the beginning. The post-action associated with reset sets the value of step to zero.

This has the effect that the abstract state ended is no longer true so that the method stepaction that was constrained, in activityWithEnd, by a condition starting that the object should not be at this state, may again be accepted.

3.2. Object coordination.

3.2.1. *Associating Audioeffects to Playback of a Video Clip.* In this example we use state notification and the dinamic definition of abstract states and state predicates to associate some audio effects to the play back a video clip.

In this example an instance of a videoPlayer class reads video frames from a file and displays them in a window on the screen. An instance of an AudioEffectManager class associates dynamically with the videoPlayer object a state predicate and a set of abstract states that "annotate" the video clip. The AudioEffectManager uses state notification to

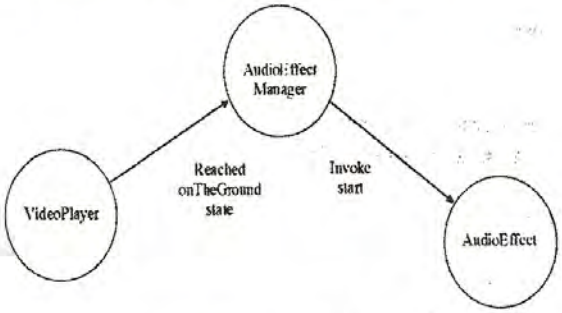

FIGURE 5. Adding audio effects to video

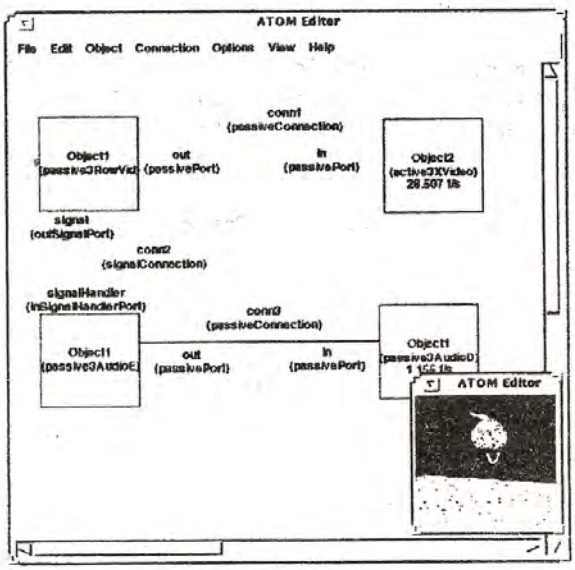

FIGURE 6. The Active Object Model Editor: Teapot example

wait for the occurrence of a visual event and activates an audioEffect object that plays the associated sound.

In this particular example the video shows a teapot that jumps, does a looping in the air and falls on the ground. A state predicate of class flightVideoAnnotations is associated with the videoPlayer object and defines the abstract states: inTheAir, onTheGround and highest. These states are associated with visual events in the playback of the video-clip. For instance, the videoPlayer object is at the abstract state onTheGround when it displays the video frames where the teapot is on the ground. In the example, the AudioEffectManager uses state notification to wait for the event that the videoPlayer is at the abstract state onTheGround to generate an impact sound. Figure 5 shows the program structure used for the example.

The state predicate, flightVideoAnnotation, that is used in this example to define abstract states that are associated with visual events is shown in Figure 7. The association

```
class flightVideoAnnotations:
  obj = None
  functs = {
    'highest': (lambda x: x.frame == 13),
    'inTheAir': (lambda x: 1 < x.frame < 26)
  }
  def __init__(self, object):
    self.obj = object
  def evalState(self, state, obj):
    return self.functs[state[0]](obj)
class AudioEffectManager(ActiveObjectSupport):
  def __init__(self, video, state, audioEffect):
    self.state = state, self.video = video
    self.audioEffect = audioEffect
    self.video.newPred(flightVideoAnnotations,
                       ['highest', 'onTheAir', 'onTheGround'],))
    self.fall = self.video.notifyRequest(('onTheGround',))
  def activity(self):
    while not self.atState(('stopped',)):
      self.suspendUntil(self.fall), self.audioEffec.start()
```

FIGURE 7

between the abstract states and visual events in the video clip is made by using the instance variable step of the videoPlayer that provides information about the frame that is displayed. The class defines three functions that establish the relationsheep between video frames and abstract states. A state predicate class such as videoflightVideoAnnotations may in fact be generated automatically by a tool that allows a user go view a video and associate interactively frames with abstract states.

The class AudioEffectManager is shown in figure. At its creation an AudioEffectManager object is acquitainted to a videoPlayer object and an AudioEffect object, using the newPred method, it associates dynamically with the VideoPlayer object the state predicate flightVideoAnnotation that defines the abstract states associated with the visual events of interest. Then, it requests by calling the method notifyRequest, to be notified when the videoPlayer object is at the abstract state onTheGround. The call to notifyRequest returns an object representing the notification event. After initialization the activity method of the AudioEffectManager is executed in a new thread. In this method the event object, returned by the call to notifyRequest, is used in the call to the suspendUntil method to suspend the execution of the method until the videoPlayer object reaches the abstract state ontheGround. When the state is reached the thread is resumed and it invokes the AudioEffect's start method to playback the audio effect.

To keep the example simple, the association of abstract states to audio effects is done statically in the code of the AudioEffectManager class that allows the association of different audio effects to several abstract states of the videoPlayer. Furthemore, the association of audio effects to abstract states can be specified in a list that is passed to the AudioEffectManager at initialization rather than having it hard-wired in the code of the class in the example.

4. Conclusion

We have presented an active object model that combines concurrency and object-oriented features. The model integrates concurrency and object-oriented features in such a way that alleviates several known problems for taking advantage of the software reuse potential of object-oriented features in the development of concurrent software. In addition, the model provides support for novel ways to combine concurrent object behaviors.

The model introduces the novel features of abstract states, state predicates and state notification that can be used to synchronize the actions of a single objects as well as coordinate the execution of sets of objects. This done in a way that is compatible with polymorphism and inheritance and provide novel ways to support reuse by combining active object behaviours.

The proposed model has been implemented as an extension to the programming language Python. We have started using the prototype for the development of multimedia programming environment based on active objects and have had very positive experiences. The implementation of the model in language that is freely available on different platforms this will allow us and other researchers to gain more experience with the model by using it for developing concurrent software and further refine the model's features.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object Interactions Using Composition Filters*, Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, ed. R. Guerraoui, O. Nierstrasz, M. Riveill.

[2] P. America, *Inheritance and Subtyping in a Parallel Object-Oriented Language*, Proceedings of the ECOOP '87, ed. J. Bezivin, J-M. Hullot, P. Cointe and H. Lieberman.

[3] C. Atkinson, S. Goldsack, A. di Maio, R. Bayan, *Object-Oriented Concurrency and Distribution in DRAGOON*, JOOP, March/April 1991.

[4] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley/ACM Press, 1991Lodewijk Bergmans, "Composing Concurrent Objects", Ph. D. Thesis, University of Twente, 1994.

[5] T. Bloom, *Evaluation Syncronization Mechanisms*, in 7th International ACM Symposium on Operating Systems Principles, 1977.

[6] D. Caromel, *Concurrency and Reusability: From Sequential to Parallel*, JOOP, Sept./Oct. 1990.

[7] S. Frolund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, Proceedings ECOOP '92, ed. O. Lehrmann Madsen.

[8] S. Frolund, G. Agha, *A language Framework for Multi-Object Coordination*, Proceedings ECOOP '93.

[9] S. Matsuoka, K. Taura, A. Yonezawa, *Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object-Oriented Languages*, Proceedings OOPSLA '93.

[10] B. Meyer, *Systematic Concurrent Object-Oriented Programming*, Communications of the ACM, Sept. 1993.

[11] C. Neusius, *Synchronisation Actions*, Proceedings of ECOOP '91, July 1991.

[12] M. Papathomas, D. Konstantas, *Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid*, in Object Management, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1990.

[13] M. Papathomas, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, Ph. D. Thezis, University of Geneva, 1992.

[14] M. Papathomas, G. S. Blair, G. Coulson, *A model for Active Object Coordination and its use for Distributed Multimedia Applications*, ECOOP '94 Workshop on Coordination Models and Languages for Parallelism and Distribution, Bologna, Italy, July 1994.

[15] M. Papathomas, *Concurrency in Object-Oriented Programming Languages*, in Object-Oriented Software Composition, Prentice Hall, O. Nierstrasz and D. Tsichritzis eds.

[16] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in Object-Oriented Concurrent Programming, ed. M. Tokoro, MIT Press, 1987.

LSR-IMAG Grenoble, France

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: scuty@cs.ubbcluj.ro

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

UNE APPROCHE SUR LA MODÉLISATION D'HÉRITAGE DANS UN SYSTÈME ORIENTÉ-OBJET

G. MOLDOVAN AND C. BOBOILĂ

Résumé. Les domaines d'applications tels que: la conception assistée par ordinateur (CAO), la production de documents incorporant textes (images et graphique), le genie logiciel, nécessitent une représentation d'objets complexes.

L'approche orienté objet offre de nombreux avantages pour modéliser ces domaines d'applications. A partir de constructeurs tels que les constructeurs de n-uplet, ensemble, tableau, liste, et en imbriquant arbitrarement ceux-ci, les systèmes orientés-objet supportent la notion d'objet complexes avec identité d'objet.

Cet article présente un modèle de définition d'objets complexes avec identité d'objet au moyen des graphes. On définit les notions de type, valeur structurée, objet, la relation de sous-typage, le graphe des types, graphe de composition d'objets et graphe d'héritage des classes pour mettre en évidence les connexions inter-objet.

1. Introduction

Les domaines tels que: la conception assistée par ordinateur (CAO), la production de documents incorporant textes, images et graphiques, le génie logiciel, nécessitent la gestion d'une grande variété de types de données ainsi que les liens entre ces données (souvent imbriquées).

Le modèle relationnel [7] ne permet pas de modéliser efficacement ces types variés de données ainsi que leur imbrication. Dans ce modèle, les données sont représentées sous forme de relations "plates", c'est la contrainte de première forme normale. Les attributs d'un n-uplet étant nécessairement des valeurs atomiqurs (entiers, caractères, réels, booléens), il est difficile de représenter un objet complexe dans son ensemble.

Les premières évolutions du modèle relationnel ont consisté à enrichir son pouvoir de modélisation. La difficulté de représenter des structures complexes par des relations a

Received by the editors: October 9, 1996.

1991 *Mathematics Subject Classification.* 68P15, 68P05, 68Q65.

1991 *CR Categories and Descriptors.* H.2.1 [Database Management]: Logical Design – data models; H.2.3 [Database Management]: Languages – data description languages; D.3.2 [Programming Languages]: Language Classifications – object-oriented languages.

conduit à l'élaboration de modèles ne se restreignant plus à de seules valeurs atomiques, mais acceptant des valeurs structurées.

Plusieurs tentatives d'extension du modèle relationnel ont été effectuées au cours de ces dernières années [4].

Le rest du papier est organisé de la manière suivante: la Section 2 décrit un modèle à objets complexes avec identité d'objet, la Section 3 présente la rélation de sous-typage pour définir le graphe des types, le graphe de composition d'objets et le graphe d'héritage des classes, nous concluons ensuite Section 4.

2. Un modèle à objets complexes avec identité d'objet

Nous avons considéré les ensembles suivants. Un ensemble fini de domaines $D_1, \ldots, D_n$, $n \geq 1$ (par exemple l'ensemble Z de nombres entiers). Notons D l'union des domaines $D_1, \ldots, D_n$. Nous supposons que les domaines sont disjoints. Un ensemble dénombrable et infini A, qui s'appelle **univers d'attributs**. Les éléments de A sont des noms pour les champs de la structure. Un ensemble dénombrable et infini I d'identificateurs. Les éléments de I seront utilisés comme d'identificateurs pour d'objets.

Les types sont récursivement construits de la manière suivante:

Définition 2.1. *(Types) Soit A un univers d'attributs et T un ensemble de types prédéfinis (***integer, real, boolean, string, char***, etc).*

(i): *Tout élément de T est un type (dit* **atomique***).*

(ii): *Si $t_1, \ldots, t_n$ sont des types et $a_1, \ldots, a_n$ des attributs de A, alors $t = [a_1 : t_1, \ldots, a_n : t_n]$ est un type de structure n-uplet.*

(iii): *Si t_1 est un type alors $t = \{t_1\}$ est un type ensemble.*

(iv): *Si t_1 est un type alors $t = (t_1)$ est un type liste.*

(v): *Notons T l'ensemble des types.*

Les valeurs sont récursivement construites de la manière suivante.

Définition 2.2. *(Valeurs) Soit A un univers d'attributs et D un domaine de valeurs atomiques. Une valeur simple est prise dans l'un de types prédéfinis que sont les entiers (***integer***), les valeurs réelles (***real***), les valeurs logiques (***boolean***), les caractères (***char***) et les chaînes de caractères (***string***).*

(i): *Tout élément de D est une valeur (dite atomique).*

(ii): *Un identificateur d'objet est une valeur.*

(iii): *Si $v_1, \ldots, v_n$ sont des valeurs et $a_1, \ldots, a_n$ des attributs de A, alors $v = [a_1 : v_1, \ldots, a_n : v_n]$ est une valeur structurée de type n-uplet.*

(iv): *Si $v_1, \ldots, v_n$ sont des valeurs distinctes alors $v = \{v_1, \ldots, v_n\}$ est une valeur structurée de type ensemble.*

(v): *Si $v_1, \ldots, v_n$ sont des valeurs alors $v = (v_1, \ldots, v_n)$est une valeur structurée de type liste.*

(vi): *Notons V l'ensemble des valeurs.*

Les objets sont construits de la manière suivante:

Définition 2.3. *(Objets) Un objet est un couple $o =< i, v >$ où i est un élément de I (un identifiant) et v est une valeur.*

3. La rélation de sous-typage

Notons $n..m$ le sous-type du type **integer** correspondant à l'ensemble des entiers de n à m bornes comprises. On définit récursivement la relation de sous-typage de la façon suivante:

Définition 3.1. *(Sous-types)*

(i): *$n..m \preceq p..q$ si et seulement si $p \leq n$ et $q \leq m$.*

(ii): *Si $t_1, \ldots, t_m, u_1, \ldots, u_n$ sont des types et $a_1, \ldots, a_m$ sont des attributs alors $[a_1 : t_1, \ldots, a_m : t_m] \preceq [a_1 : u_1, \ldots, a_n : u_n]$ si et seulement si $t_i \preceq u_i$ pour tout i de 1 à n et $n \leq m$.*

(iii): *Si t_1, t_2 sont des types alors $\{t_1\} \preceq \{t_2\}$ si et seulement si $t_1 \preceq t_2$.*

(iv): *Si t_1, t_2 sont des types alors $(t_1) \preceq (t_2)$ si et seulement si $t_1 \preceq t_2$.*

Définition 3.2. *(Classes) Une classe est un couple $C =< O, T, M >$ où O est l'ensemble d'objets (instances) de la classe (***extension** *de la classe), T est le type d'objets de la classe, et M, l'ensemble de méthodes appliquables sur l'objets de la classe.*

Le type T, associé à une classe C reflète la hiérarchie de composition des objets de cette classe, c'est-à-dire les liens que possèdent ceux-ci avec d'autres objets [2, 5, 6].

Le type T peut être représenté par un graphe orienté étiqueté, dont la définition suit:

Définition 3.3. *(Graphe de type GTC)*

(i): *$GTC = (N_T, E_T)$ où:*

(ii): *N_T est l'ensemble des sommets étiquetés. Chaque sommet représente un type et est étiqueté au moyen de $\beta : N_T \to T$*

(iii): *Si t est un type atomique alors $t \in N_T$ et $\beta(t) = t$.*

(iv): *Si $t_1, \ldots, t_n \in N_T$ et $t = [a_1 : t_1, \ldots, a_n : t_n]$ alors $t \in N_T$ et $\beta(t) = []$.*

(v): *Si $t_1 \in N_T$ et $t = \{t_1\}$ alors $t \in N_T$ et $\beta(t) = \{\}$.*

(vi): *Si $t_1 \in N_T$ et $t = (t_1)$ alors $t \in N_T$ et $\beta(t) = ()$.*

(vii): *E_T est l'ensemble des arcs orientés et étiquetés au moyen de $\gamma : E_t \to A$ où A est l'ensemble des noms d'attributs.*

(viii): *Si $t = [a_1 : t_1, \ldots, a_n : t_n]$ alors $(t, t_i) \in E_T$ et $\gamma(t, t_i) = a_i$ pour tout i de 1 à n.*

(ix): *Si $t = \{t_1\}$ alors $(t, t_1) \in E_T$ et $\gamma(t, t_1)$ n'est pas définie.*

(x): *Si $t = (t_1)$ alors $(t, t_1) \in E_T$ et $\gamma(t, t_1)$ n'est pas définie.*

La totalité des liens entre tous les types présents dans un système S est donnée par le **Graphe des Types GTS**.

Définition 3.4. *(Graphe des types GTS) Soit S, l'ensemble des classes du système. Le graphe des types GT est défini comme suit: $GTS = \cup_{T \in S} GTC = (\cup_T N_T, \cup_T E_T)$.*

Etant donné l'ensemble O d'objets de la classe C, les liens entre ces objets pourront être représentés par un graphe orienté. Nous appelons ce graphe: **Graphe de Composition d'Objets** *GCO*. La définition d'un tel graphe suit:

Définition 3.5. *(Graphe de composition d'objets GCO) Soit I un ensemble d'identificateurs d'objets et V l'ensemble de valeurs atomiques ou structurées. Le GCO est défini pour les objets/valeurs par:*

(i): *$G_I = (V_I, E_I)$ où:*

(ii): *V_I est l'ensemble des nœuds chaque nœud représente une valeur et est étiqueté par $\alpha : V_I \to I$ et $\beta : V_I \to V$.*

(iii): *Si v est associée à l'objet identifié par i alors $v \in V_I$ et $\alpha(v) = i$.*

(iv): *Si v est une valeur atomique alors $v \in V_I$ et $\beta(v) = v$.*

(v): *Si $v_1, \ldots, v_n \in V_I$ et $v = [a_1 : v_1, \ldots, a_n : v_n]$ alors $v \in V_I$ et $\beta(v) = []$.*

(vi): *Si $v_1, \ldots, v_n \in V_I$ et $v = \{v_1, \ldots, v_n\}$ alors $v \in V_I$ et $\beta(v) = \{\}$.*

(vii): *Si $v_1 v_n \in V_I$ et $v = (v_1, \ldots, v_n)$ alors $v \in V_I$ et $\beta(v) = ()$.*

(viii): *E_I est l'ensemble des arcs étiquetés au moyen de $\gamma : E_I \to A$, où A est l'ensemble des noms d'attributs.*

(ix): *Si* $v = [a_1 : v_1, \ldots, a_n : v_n]$ *alors* $(v, v_k) \in E_I$ *et* $\gamma(v, v_k) = a_k$ *, pour tout* k *de* 1 *à* n.

(x): *Si* $v = \{v_1, \ldots, v_n\}$ *alors* $(v, v_k) \in E_I$ *et* $\gamma(v, v_k)$ *n'est pas définie, pour tout* k *de* 1 *à* n.

(xi): *Si* $v = (v_1, \ldots, v_n)$ *alors* $(v, v_k) \in E_I$ *et* $\gamma(v, v_k)$ *n'est pas définie, pour tout* k *de* 1 *à* n.

Le *GCO* peut être vu comme une "instanciation" du graphe des types *GT*. La relation d'héritage entre classes peut être représentée par une relation de sous-typage entre types.

Définition 3.6. *(La relation d'héritage des classes) Soit* C *l'ensemble des toutes les classes et* $C_1 =< O_1, T_1, M_1 >$, $C_2 =< O_2, T_2, M_2 >$, *deux classes de* C. $C_2 \preceq C_1$ *si et seulement si,* $T_2 \preceq T_1$ *et* $M_1 \subseteq M_2$.

Si deux classes sont en relation d'héritage (une relation d'ordre partiel), on appelle C_1la super-classe de C_2 et C_2 la sous-classe de C_1.

Définition 3.7. *(Le graphe d'héritage des clases GHC)*

(i): $G = (X, U) = (X, \Gamma)$ *où:* X *est l'ensemble des sommets, ou classes, et* U *la relation d'heritage* ($\preceq$), *un ensemble de* $X \times X$. G *est muni d'une racine, notée* C_0.

(ii): Γ *est la relation multi-valuée des successeurs d'un sommet, équivalente à* U. *On utilisera aussi* Γ^{-1} *qui représente les prédécesseurs.*

On note $\lambda = (C_0, C_1, \ldots, C_n)$ *et on appelle* λ *un chemin de* G *si et seulement si* $\{(C_0, C_1), (C_1, C_2), ..., (C_{n-1}, C_n)\} \subseteq U$.

Définition 3.8. *(Sous-graphe de descendants) Soit* C_r *un sommet du graphe d'héritage.* $G_1 = (X_1, U_1)$ *où:* $X_1 = \{C_X \in X \mid \exists \lambda = (C_0, \ldots, C_X)$ *un chemin dans* $G\}$. C_1 *est la racine de sous-graphe* G.

4. Conclusion

Nous avons présenté ici les concepts d'un modèle à objets complexes avec identité. Dans les SGBDOO le concept d'objet est fondamental. Nous avons défini le concept de type complexe à partir de types atomiques et de constructeurs qui s'appliquent récursivement.

La relation de sous-typage entre types peut être considérée comme une possibilitée d'établir une hiérarchie d'héritage sur l'ensemble des classes.

On représent souvent les liens entre types et objets au moyen d'un graphe. Nous avons construit le graphe des types GT, le graphe de composition d'objets GCO, et le graphe d'héritage GHC.

Le graphe des types, le graphe de composition d'objets et le graphe d'héritage jouent un rôle prépondérant dans la mise en œuvre des stratégies de regroupement d'objets sur disque dans un système de bases de données orienté-objet [5, 6].

Bibliographie

[1] S. Abiteboul, P. Kanellakis, *Object Identity as Query Language Primitive*, In Proc. of the ACM SIGACT- SIGMOD Symp. on Principles of Database Systems, Juin, 1989.

[2] M. Adiba, C. Collet, *Objets et Bases de Données. Le SGBD O2*, Hermès, Paris, 1993.

[3] M. Atkinson, P. Buneman, *Types and Persistence in Database Programming Languages*, ACM Computing Surveys, June, 1987.

[4] A. Bancilhon, S. Khoshafian, *A calculus for Complex Objects*, ACM PDDS Conference 1986.

[5] V. Benzaken, *Regroupement d'objets sur disque dans un système de bases de données orienté-objet*, Thèse de doctorat, Paris, 1990.

[6] V. Benzaken, A. Doucet, *Bases de Données orientées objet. Origines et principes*, Armand Colin, 1993.

[7] E.F. Codd, *A Relational Model of Data for Large Shared Data Banks*, CACM Vol 13 No 6, June, 1970.

[8] M. Humbert, *Les Bases de Données*, Hermès, 1989.

[9] S. Khoshafian, G. Copeland, *Object Identity*, OOPSLA 86, Portland Oregon, Sept. 1986.

[10] C. Lecluse, P. Richard, F. Velez, *O2, an Object- Oriented Data Model*, ACM 3/1988.

[11] C. Lecluse, P. Richard, *Modeling Complex Structures in Object-Oriented Databases*, Proc. of the ACM PODS Conference, Philadelphie, 1989.

[12] GR. Moldovan, *Modelul Relational pentru Baze de Date, Metodologii si tehnici moderne de proiectare si scriere a programelor*, Bucuresti, 1981, pag. 213-226.

[13] P. Richard, *Des Objets Complexes aux Bases de Données Orientées - objets*, Thèse de doctorat, Paris, 1991.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: moldovan@cs.ubbcluj.ro

University of Craiova, Faculty of Mathematics – Computer Science, RO-1100 Craiova, Romania

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

COMPILING DEFINITE CLAUSE GRAMMARS

D. TĂTAR

Abstract. The original motivation for Chomsky's phrase structure grammar was the description of natural languages (NL). The most frequently used of them, CFG's, are not in general regarded as adequate for all the aspects that occurs in NL processing. A generalisation of CFG's, definite clause grammars (DCG) is better: in this paper we introduce some methods of the association of a DCG to a CFG and the connections between the queries addressed to first and the language generated by the second.

1. Introduction

It is very well known how a CFG can describe a subset of a NL, as in the following example

Example 1.1. *Sample grammar 1.*

$G = (I_N, I_T, S, P)$

$I_N = \{sentence, verb, noun - phrase, article, adjective, noun, preposition\}$,

$I_T = \{'give','the','a','best','last','definition','notion','of'\}$,

$S = sentence$,

$P = \{sentence \to verb, noun - phrase,$

$noun - phrase \to article, adjective, noun,$

$noun - phrase \to article, adjective, noun, preposition, noun - phrase$

$verb \to' give'$

$article \to' a'$

$article \to' the'$

Received by the editors: September 21, 1996.

1991 *Mathematics Subject Classification.* 68N17, 68S05.

1991 *CR Categories and Descriptors.* D.1.6 [Programming Techniques]: Logic programming; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic; I.2.7 [Artificial Intelligence]: Natural Language Processing.

$$adjective \rightarrow' best'$$
$$adjective \rightarrow' last'$$
$$noun \rightarrow' definition'$$
$$noun \rightarrow' notion'$$
$$preposition \rightarrow' of'\}$$

The production rules allow us to verify that the string:

$$'give','the','best','definition','of','the','last','notion'$$

is a legal sentence of CFG in this sample.

Let us, on the other hand, regard a Horn clause in a definite program:

$$h : -a_1, a_2, \cdots, a_n$$

or

$$h \leftarrow a_1, a_2, \cdots, a_n$$

We read the implication $h \leftarrow a_1, a_2, \cdots, a_n$ as a production rule : $h \rightarrow a_1, a_2, \cdots, a_n$ which is interpreted as "*h is well-formed if* " $a_1, a_2, \cdots, a_n$ *is well-formed* and such a production rule corresponds to each Horn clause.

A first apparent difference between a production rule and a clause is the order in which the element of the right-hand part of the production rule: the order of elements in the body of a clause is immaterial, due to the commutativity of the conjunction operation. In fact, the precise order of elements in the body of a clause is pushed in the precise order of elements of a list.

We will define a first type of DCG associated with a CFG, and we will call them "definite clause grammar of first type" (DCG-FT).

Definition 1.2. *Let $G = (I_N, I_T, S, P)$ a CFG for a sequence of NL, as in sample 1, where I_N is a set of syntactic categories and I_T is a set of words terminal. A DCG-FT associated with G is a* **definite program** *P, defined as follows:*

- *for each non-terminal X in I_N there exists a predicate p_X with one argument such that $p_X(Y)$ succeeds iff Y is a list of syntactic category X.*
- *for each production rule $X \rightarrow X_1, X_2 \cdots X_n$ there exists a Horn clause :*

$$p_X([Y_1|Y_2|\cdots Y_n]) : -p_{X_1}(Y_1), \cdots, p_{X_n}(Y_n)$$

- *for each production rule* $X \to t, t \in I_T$ *there exists a Horn clause* $p_X([t])$.

In this condition, the following theorem is proved.

Theorem 1.3. *Let* $G = (I_N, I_T, S, P)$ *be a CFG. A word* $w = w_1, w_2, \cdots, w_n \in L(G)$, *where* $w_i \in I_T$,$i = 1, \cdots, n$, *iff*

$$p_S([w_1, w_2, \cdots, w_n])$$

succeeds in DCG-FT, associated with G as in above definition.

Proof. We will prove by induction on the length of the deduction a more general result:

For $\forall X$, $X \Rightarrow *w_1, w_2, \cdots, w_n$, where $w_i \in I_T$,$i = 1, \cdots, n$, iff $p_X([w_1, w_2, \cdots, w_n])$ *succeeds* in DCG-FT, associated with G as in above definition.

If the length of the deduction is 1, then it is $X \to t$, and by definition $p_X([t])$ succeds. Let us suppose that $\forall X$, $X \Rightarrow *w_1, w_2, \cdots, w_n$, is a deduction of length $k+1$.This deduction is of the form

$$X \to X_1, X_2 \cdots X_m \Rightarrow *w_1, \cdots, w_n$$

Then, there exist the lists $L_1, L_2, \cdots, L_m$ such that:

$$1.[w_1, w_2, \cdots, w_n] = [L_1|L_2|\cdots|L_m]$$

and

$$2.X_i \Rightarrow *L_i, i = 1, \cdots, m$$

are the deductions of length $\leq k$. By induction assumption $p_{X_i}(L_i)$ succeds, $i = 1, \ldots, m$.

Since for the rule

$$X \to X_1, X_2 \cdots X_m$$

there exists a Horn clause : $p_X([Y_1|Y_2|\cdots|Y_m]) : -p_{X_1}(Y_1), \cdots, p_{X_m}(Y_m)$ and $p_{X_i}(L_i)$ succeds, $i = 1, \cdots, m$, then that $p_X([L_1|\cdots|L_m])$ succeds too.

As $[w_1, w_2, \cdots, w_n] = [L1|L2|\cdots|Lm]$, that means that $p_X([w_1, \cdots, w_n])$ succeds.

For the converse implication, we will proceed by induction on the length n of the argument list. If $n = 1$ than $p_X([w_1])$ succeeds and, by above definition, $X \to w_1$. Let us suppose that

$$p_X([w_1, w_2, \cdots, w_n])$$

succeeds. Accordingly with the above definition, there exists a Horn clause

$$p_X([Y_1|Y_2|\cdots Y_m]) : -p_{X_1}(Y_1), \cdots, p_{X_m}(Y_m)$$

and $[w_1, \cdots, w_n]$ is obtained by concatenation of an instantiation of the list $[Y_1|\cdots|Y_m]$ .Let $[L_1|\cdots|L_m]$ be this instantiation. As $p_{X_i}(L_i)$ succeeds, $i = 1, \ldots, m$,and the length of L_i is $< n$, by induction assumption we can say that $X_i \Rightarrow *L_i$,$i = 1, \ldots, m$.

As $X \to X_1, X_2 \cdots X_m$, then $X \to X_1, X_2 \cdots X_m \Rightarrow *L_1, \cdots, L_m = w_1, \cdots, w_n$ □

Example 1.4. *The DCG-FT associated with the sample grammar 1 is the following:*

```
domains
  lista=symbol*
predicates
  sentence(lista)
  verb(lista)
  noun_ph1(lista)
  noun_ph2(lista)
  article(lista)
  adjective(lista)
  noun(lista)
  prep(lista)
clauses
  sentence([VE|NP]):-verb([VE]),noun_ph1(NP).
  sentence([VE|NP]):-verb([VE]),noun_ph2(NP).
  noun_ph1([AR,AD,NO]):-article([AR]),
                        adjective([AD]),
                        noun([NO]).
  noun_ph2([AR,AD,NO,PP|NP]):-article([AR]),
                        adjective([AD]),
                        noun([NO]),
                        prep([PP]),
                        noun_ph1(NP).
  verb([give]).
  article([the]).
```

```
adjective([best]).
adjective([last]).
noun([definition]).
noun([notion]).
prep([of]).
```

If the goal addressed to this DCG is : $sentence(X)$,then we will obtain: $X = [give, the, best, definition, of, the, last, notion\ ]$ and others 19 solutions. In the set of solution we find the following: $[give, the, best, notion, of, the, last, definition]$, which is, surely, not semantically correct. In this point another tool that is introduced by DCG can be useful, namely the " procedure calls". [3]. These are some relations between the arguments of symbols from I_N. The translation of this condition is easily realized by a DP,where the relational operators are permitted. Another generalisation realised by a DCG, apart from CFG, is the possibility of realisation of the number and the person agreement.

Example 1.5. *Sample grammar 2.*

$$sentence \rightarrow pronoun, verb, article, noun$$
$$pronoun \rightarrow' I'$$
$$pronoun \rightarrow' you'$$
$$pronoun \rightarrow' he'$$
$$pronoun \rightarrow' we'$$
$$pronoun \rightarrow' you'$$
$$pronoun \rightarrow' they'$$
$$article \rightarrow' a'$$
$$noun \rightarrow' teacher'$$
$$noun \rightarrow' teachers'$$
$$verb \rightarrow' am'$$
$$verb \rightarrow' are'$$
$$verb \rightarrow' is'$$
$$verb \rightarrow' are'$$

In this CFG grammar we can obtain the following incorrect sentence: $'We', 'is', 'the', 'teachers'$.

Let us transform CF production rules as:

$$sentence(X,Y) \rightarrow pronoun(X,Y), verb(X,Y), article, noun(X).$$

$$pronoun(1,1) \rightarrow' I'$$

$$pronoun(1,2) \rightarrow' you'$$

$$pronoun(1,3) \rightarrow' he'$$

$$pronoun(2,1) \rightarrow' we'$$

$$pronoun(2,2) \rightarrow' you'$$

$$pronoun(2,3) \rightarrow' they'$$

$$article \rightarrow' a'$$

$$noun(1) \rightarrow' teacher'$$

$$noun(2) \rightarrow teachers'$$

$$verb(1,1) \rightarrow' am'$$

$$verb(1,2) \rightarrow' are'$$

$$verb(1,3) \rightarrow' is'$$

$$verb(2,1) \rightarrow' are'$$

$$verb(2,2) \rightarrow' are'$$

$$verb(2,3) \rightarrow' are'$$

In this DCG grammar, the sentence as above cannot be obtained. For example, we can obtain the following correct sentence:

$$sentence(1,1) \rightarrow pronoun(1,1), verb(1,1), article, noun(1) \Rightarrow *'I','am','a','teacher'$$

A rewriting of a word in this DCG grammar is obtained by a double process: an application of the relation $\Rightarrow *$ as in the usual definition and a *unification* process. This is a common fact with the *unification grammar* as in [1].

The DCG-FT obtained as in the above procedure is the following:

```
domains
 lista=symbol*
predicates
 sentence(integer,integer,lista)
 pronoun(integer,integer,lista)
 verb(integer,integer,lista)
 noun(integer,lista)
```

```
 article(lista)
clauses
 sentence(X,Y,[Pn,V,Ar,N]):-pronoun(X,Y,[Pn]),
                            verb(X,Y,[V]),
                            article([Ar]),
                            noun(X,[N]).
 pronoun(1,1,[i]).
 pronoun(1,2,[you]).
 pronoun(1,3,[he]).
 pronoun(2,1,[we]).
 pronoun(2,2,[you]).
 pronoun(2,3,[they]).
 article([a]).
 noun(1,[teacher]).
 noun(2,[teachers]).
 verb(1,1,[am]).
 verb(1,2,[are]).
 verb(1,3,[is]).
 verb(2,1,[are]).
 verb(2,2,[are]).
 verb(2,3,[are]).
```

The goal: $sentence(1,1,X)$ addressed to this DCG-FT obtains,
$X = ['I','am','a', teacher']$.

2. Obtaining the parse tree

If we want to obtain the syntactic tree (the parse tree) of a sentence,we must modify the rules of association of DCG to a CFG.

Let us consider the following sentence in the Sample grammar 1:

$$'give','the','best', definition','of','the','last','notion'$$

The above sentence can be obtained by the concatenation of all the leaves from the parse tree, from left to right. This tree can also be represented by means of the below

construction, which suggests better the process of generation of a word:

$$sentence(v('give'), np(art('the'), adj('best'), n('definition'),$$
$$prep('of'), np(art('the'), adj('last'), n('notion'))))$$

We will call such a tree an *annotated* tree. It is obtained by reading the tree top-down, such that each production rule $X \to X_1, X_2, \cdots, X_n$, where $s_X, s_{X_1}, s_{X_2}, \cdots, s_{X_n}$ are syntactic categories of $X_1, X_2, \cdots, X_n$, is *annotated* with $s_X(s_{X_1}, \cdots, s_{X_n})$.

A DCG-FT which obtains the syntactic trees is defined as follows.

Definition 2.1. *Let $G = (I_N, I_T, S, P)$ a CFG for a sequence of NL. A DCG-FT associated with G for the syntactic tree is a definite program P defined as follows:*

- *for each non-terminal X in I_N which occurs in a production rule $X \to X_1, \cdots, X_n$ and which is of the syntactic category s_X, there exists a predicate*

$$p_X(s_X(V_1, \cdots, V_n)), [V_1', \cdots, V_n'])$$

such that $p_X(Y, Z)$ succeeds iff Y is of the syntactic category of p_X and Z is a list of the syntactic categories of its descendents.

- *for each production rule $X \to X_1, X_2 \cdots X_n$ there exists a Horn clause:*

$$p_X(s_X(V_1, V_2, \cdots V_n), [V_1', V_2', \cdots, V_n']) : -p_{X_1}(V_1, [V_1']), \cdots, p_{X_n}(V_n, [V_n'])$$

- *for each production rule $X \to t, t \in I_T$ there exists a Horn clause*

$$p_X(s_X(t), [t]).$$

In this case, the following theorem is proved.

Theorem 2.2. *Let $G = (I_N, I_T, S, P)$ be a CFG . A word $w = w_1, w_2, \ldots, w_n \in L(G)$, where $w_i \in I_T$,$i = 1, \ldots, n$, iff $p_S(A, [w_1, w_2, \cdots, w_n])$ succeeds in DCG-FT for the syntactic tree , associated with G as in the above definition,with A being the annotated tree of w.*

The DCG-FT for the syntactic tree, for CFG in *Sample grammar 1*, obtained as in the above procedure, is the following:

```
domains
  dom=v(symbol);art(symbol);adj(symbol);no(symbol);
  np(dom,dom,dom);sn(dom,dom);
```

```
  npc(dom,dom,dom,dom,dom);p(symbol)
  lists=symbol*
predicates
  sentencet(dom,lists)
  verbt(dom,lists)
  noun_pht1(dom,lists)
  noun_pht2(dom,lists)
  articlet(dom,lists)
  adjectivet(dom,lists)
  nount(dom,lists)
  prept(dom,lists)
clauses
  sentencet(sn(VE,NP),[V|N]):-verbt(VE,[V]),noun_pht1(NP,N).
  sentencet(sn(VE,NP),[V|N]):-verbt(VE,[V]),noun_pht2(NP,N).
  noun_pht1(npc(AR,AD,NO,PP,NP),[Art,Adj,Nn,Prep|Nph]):-
                        articlet(AR,[Art]),
                        adjectivet(AD,[Adj]),
                        nount(NO,[Nn]),
                        prept(PP,[Prep]),
                        noun_pht2(NP,Nph).
 noun_pht2(np(AR,AD,NO),[Art,Adj,N]):-articlet(AR,[Art]),
                        adjectivet(AD,[Adj]),
                        nount(NO,[N]).
  verbt(v(give),[give]).
  articlet(art(the),[the]).
  adjectivet(adj(best),[best]).
  adjectivet(adj(last),[last]).
  nount(no(definition),[definition]).
  nount(no(notion),[notion]).
  prept(p(of),[of]).
```

For the goal: *sentence*(X, Y) this program obtains 20 solutions, in which Y is a correct sentence, and X is the *annotated* syntactic tree associated with it.

3. Definite clause grammar of second type

Another sort of DCG associated with a CFG, which can describe a syntactic tree of a sentence, if it is correct, is the DCG of the second type (DCG-ST) obtained from the below procedure.

Definition 3.1. *Let $G = (I_N, I_T, S, P)$ a CFG . A DCG-ST associated with G is a definite program P defined as follows:*

- *for each non-terminal X in I_N there exists a predicate p_X with two arguments such that $p_X(Y, Z)$ succeeds iff Y is a list whose head is of the syntactic category X, and whose tail is the list Z.*
- *for each production rule $X \to X_1, X_2 \cdots X_n$ there exists a Horn clause:*

$$p_X(Y, Z) : -p_{X_1}(Y, Y_1), \cdots, p_{X_{n-1}}(Y_{n-2}, Y_{n-1}), p_{X_n}(Y_{n-1}, Z)$$

- *for each production rule $X \to t, t \in I_T$ there exists a Horn clause*

$$p_X(Y, Z) : -Y = [t|Z]).$$

The following theorem can be proved.

Theorem 3.2. *Let $G = (I_N, I_T, S, P)$ be a CFG. A word $w = w_1, w_2, \ldots, w_n \in L(G)$, where $w_i \in I_T$,$i = 1, \cdots, n$, iff*

$$p_S([w_1, w_2, \cdots, w_n], [])$$

succeeds in DCG-ST, associated with G as in the above definition.

In the following we will build the DCG-ST associated with the sample grammar 3.

Example 3.3. *Sample grammar 3.*

$$sentence \rightarrow subject, verb, complement$$
$$sentence \rightarrow article, noun$$
$$complement \rightarrow adjective$$
$$article \rightarrow ['the']$$
$$noun \rightarrow ['work']$$
$$noun \rightarrow ['child']$$
$$adjective \rightarrow ['nice']$$
$$adjective \rightarrow ['sage']$$
$$verb \rightarrow ['is']$$

```
 lista=symbol*
predicates
 sentence(lista,lista)
 subject(lista,lista)
 verb(lista,lista)
 complement(lista,lista)
 article(lista,lista)
 adjective(lista,lista)
 noun(lista,lista)
clauses
 sentence(Y,Z):-subject(Y,Y1),
                verb(Y1,Y2),
                complement(Y2,Z).
 subject(Y,Z):-article(Y,Y1),noun(Y1,Z).
 complement(Y,Z):-adjective(Y,Z).
 article(Y,Z):-Y=[the|Z].
 noun(Y,Z):-Y=[work|Z].
 noun(Y,Z):-Y=[child|Z].
 verb(Y,Z):-Y=[is|Z].
 adjective(Y,Z):-Y=[nice|Z].
```

If the goal is: $sentence(Y, [])$, then the solution is: $Y = ['the','child','is','nice']$

References

[1] M. Johnson, *Attribute-value logic and the theory of grammar*, CSLI, 1988.

[2] D. Tatar, *Logic grammars as formal languages*, Studia Universitas "Babes-Bolyai", Mathematica, XXXIX (1994), pp. 75-82.

[3] A. Thayse (ed), *From standard logic to logic programming*, John Wiley & Sons, 1988.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: dtatar@cs.ubbcluj.ro

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

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

SAADI: SOFTWARE FOR FUZZY CLUSTERING AND RELATED FIELDS

H. F. POP

Abstract. The article describes a software product whose main task is mainly to perform automatical classification of different data sets. The system, programmed by the author, implements both traditional and original algorithms [3]. For each of its components the implemented algorithms and details regarding its functioning are specified. Being programmed by using modern techniques (Object Oriented Programming, Windows Programming), with the help of *Borland Delphi 1.0* programming environment for Microsoft Windows operating system, the product developed here proves itself to be extremely flexible, performant and with a user-friendly interface.

1. Introduction

The aim of this paper is to describe the SAADI software (System for Automatical Analysis of Data and for their Interpretation), produced with the aim of helping the research developed in Fuzzy Clustering and in different conex fields.

This software system has been realized with the help of the Windows programming facilities provided by *Borland Delphi 1.0*. All the interdependency mechanisms of the system have been programmed using object oriented programming techniques.

The SAADI system performs the following tasks:

- essential characteristics selection of a data set;
- bidimensional space projection of the data, for a better vizualization of them;
- unsupervised hierarchical, non-hierarchical and simultaneous clustering of a data set; for the horizontal unsupervised classifier we are able to determine the fuzzy set associated to a classical data set (as particular case, fuzzy regression);
- computation of the fuzzy set corresponding to a classical set and to a certain index; the version of this algorithm that uses linear prototypes has been called the Fuzzy Regression Algorithm;

Received by the editors: September 15, 1996.

1991 *Mathematics Subject Classification.* 68T35, 62H30.

1991 *CR Categories and Descriptors.* I.5.3 [Pattern Recognition] Clustering – algorithms, similarity measures.

- unsupervised hierarchical, non-hierarchical or simultaneous clustering of the characteristics set;
- supervised clustering (based on training), either with classical decision (and a separation hyperplane is produced), or with fuzzy decision (and the membership degrees of the extra point are directly produced);
- some other interesting facilities such as:
  - producing normalized data;
  - constructing significant variables for the fuzzy regression line;
  - test of the algorithm for computing the eigenvalues and eigenvectors of a square, symmetrical and positively defined matrix;
  - testing and drawing a separation hyperplane;
  - a text editor and a text viewer for data files and for results files;

The data set and, if this is the case, the initial fuzzy partition are read by the system through some ASCII files. The other variables and different clustering options are introduced by the menu system and are saved in a configuration file, so that when reruning the program they are automatically read from this file. The results are presented as a ASCII file with all the necessary information so that the user may have a clear idea concerning the operation that just took place.

2. Characteristics selector

This part of the system performs a transformation of the data from the original space in a space having fewer characteristics. Thus, it is possible to realize both a projection of the data into a reduced dimension space, i.e. combining the initial characteristics into a smaller number of new characteristics, and a selection of the most relevant characteristics out of the original ones.

Two methods are available, as follows:

- the Karhunen-Loewe method, based on the principal component analysis. There we may either project the data on the eigenvectors of the covariance matrix, eigenvectors corresponding to the greatest eigenvalues, or a characteristics selection, by considering those characteristics the nearest of the eigenvectors;
- a method proposed by Dumitrescu, based on the computation of a certain importance factor, associated to each original characteristic.

3. Bidimensional space projector

This component realises the data projection in a bidimensional space, in order to allow a good graphical visualization of data. Each of the methods used allows the displaying of the projected data both on a text display and on a graphical display (in this case the resolution is much better).

Two methods are available, as follows:

- the Karhunen-Loewe method;
- the Sammon method, based on the minimization of some criterion function built such that it is targeted the conservation as much as possible the distances between points before and after the projection.

4. Unsupervised hierarchic classifier

This component performs divisive hierarchical classification based on fuzzy sets. There are operational hierarchical classifiers based on:

- point prototypes;
- use of adaptive metric;
- two types of ellipsoidal prototypes;
- linear prototypes;
- convex combination between the point and line prototypes;
- adaptive prototypes;
- classical partition prototypes.

This type of classifiers allow the user to set some working options as:

- the type of initial partition wanted for the classification process (random, or predefined in a certain way);
- whether data normalization is intended;
- whether the graphic variant of the classifier is to be used;
- the polarization threshold beginning with which a fuzzy set is no more split;
- the error threshold beginning with which two fuzzy partitions are considered identical.

5. Unsupervised horizontal classifier

This component performs horizontal classification based on fuzzy sets. There are operational horizontal classifiers based on:

- point prototypes;
- use of adaptive metric;
- two types of ellipsoidal prototypes;
- linear prototypes;
- convex combination between the point and line prototypes;
- adaptive prototypes;
- classical partition prototypes.

This type of classifiers allows the user to set some working options as:

- the number of classes the initial data set is to be split in; if this number is equal to 1, the classifier will determine the fuzzy set associated to the given data set and to a membership threshold to be set by the user (see Section 7);
- the type of initial partition wanted for the classification process (read from outside; random, or predefined in a certain way; if the initial partition is read from outside the system will try to improve its quality by refining it);
- whether data normalization is intended;
- whether the graphic variant of the classifier is to be used;
- the error threshold beginning with which two fuzzy partitions are considered identical.

6. Unsupervised cross-classifier

This component performs hierarchical cross-classification based on fuzzy sets. Now there are operational a few variations of the hierarchical cross-classifier based on point prototypes (see [8]).

This classifier allows the user to set some working options as:

- the type of initial partition wanted for the classification process (random, or predefined in a certain way);
- whether data normalization is intended;
- whether the graphic variant of the classifier is to be used;
- the polarization threshold beginning with which a fuzzy set is no more split;

- the error threshold beginning with which two fuzzy partitions are considered identical.

7. The fuzzification component

This component aims to produce the fuzzy set corresponding to a classical set and to a certain fuzzification index [5, 7]. Currently this components works in two different ways:

- it produces a single fuzzy set corresponding to the given classical set and to the fuzzification index set by the user;
- it produces a whole family of fuzzy sets corresponding to the given classical set and to the fuzzification index taking the values 0.01, 0.02, ..., 0.98 and 0.99.

There are operational fuzzyfication components based on:

- point prototypes;
- use of adaptive metric;
- two types of ellipsoidal prototypes;
- linear prototypes;
- convex combination between the point and line prototypes;
- adaptive prototypes;
- classical partition prototypes.

This type of components allows the user to set some working options as:

- the value of the fuzzification index, in the interval $(0, 1)$;
- the type of initial partition wanted for the classification process (read from outside; random, or predefined in a certain way; if the initial partition is read from outside the system will try to improve its quality by refining it);
- whether data normalization is intended.

8. Regression algorithms

This component of the system is meant to implement different regression techniques, together with different methods for the evaluation of their quality.

There are currently available:

- the Fuzzy Regression Algorithm [7];

- a version of the algorithm above which produces a whole family of regression lines, corresponding to regression indices taking the values 0.01, 0.02, ... , 0.98 and 0.99;
- component for evaluating the quality of different regression lines.

This part of the system will soon be updated, as we intend to implement different other regression techniques.

9. Unsupervised characteristics classifier

The system allows not only the classification of the data set, but also of the characteristics set. By selecting this option, the classifiers implemented here will classify the characteristics set. In this way we are able to develop useful conclusions with respect to the interdependency of different characteristics and, eventually, we may reduce the data dimensionality.

10. Classical decision supervised classifier

This component of the system implements many training algorithms based both on classical sets and on fuzzy sets.

There are operational the following supervised classifiers:

- the Rerceptron algorithm;
- the Gallant Pocket algorithm;
- the Keller-Hunt algorithm;
- a variation of the Keller-Hunt algorithm in which the memberships are read from an input file and not postulated by the algorithm;
- the Relaxation algorithm;
- the Widrow-Hoff algorithm;
- the Ho-Kashyap algorithm;
- a variation of the Ho-Kashyap algorithm in which the computation of the inverse of a certain matrix is replaced by the use of a symmetrical positively defined matrix.

This type of classifiers allow the user to set different working options, such as:

- wether or not the classical or fuzzy set version of the classifier is requested;
- wether or not the graphical version of the classifier is requested;

- the width of the threshold interval of the membership degrees near 0.5; the learning vectors erroneously classified but with the membership degrees inside this interval will not be taken into account when making a correction of the separation vector;
- the width of the threshold interval of the product $v^T z$; the learning vectors z erroneously classified will not be taken into account when making a correction of the separation vector v if the product $v^T z$ is inside this interval;
- the initialization modality of the separation vector; the vector normal on the mediator hyperplane of the segment made by the prototypes of the two classes, or a vector having all the components equal to one another and equal to a certain given number.

11. Fuzzy decision supervised classifier

This component of the system implements fuzzy decision supervised classifiers. As compared with the classical decision supervised classifiers, that produce a separation hyperplane, this class of algorithms produce the membership degrees of the tested vector.

There are operational supervised classifiers based on [4]:

- fuzzy version of the algorithm of the nearest k neighbours;
- fuzzy version of the algorithm of the nearest prototype;
- the Restricted Fuzzy n-Means algorithm.

In this moment, all these algorithms suppose the existence of point prototypes.

12. Other components of the system

The system allows the user to test some of the algorithms used at the implementation of different classification techniques. Moreover, there are shown some extensions considered to be necessary for the good working of the system:

- component to set the directory with the data file;
- component to set the names of the files with different necessary data;
- component to set the names of the final report files, with the results produced by the system;

- components to set different working options for the supervised and unsupervised classifiers; the settings done here are immediately saved in a configuration file and will be loaded the next time the system is run;
- component to test the algorithm that computes the eigenvalues and the eigenvectors of a symmetrical and positively defined square matrix;
- component to testing and drawing a separation vector;
- component to normalize the data;
- component to view the ASCII files with data or results produced by this system; here is allowed the viewing of lines both unwrapped and wrapped at every 80 character, so that the file should be completely displayed on the screen;
- component to edit the ASCII file with data or results produced by this system.

13. Objects hierarchy of the system

In order to notice better the interdependency and the relationships between different oblects we show in Table 1 the objects hierarchy as well as a short description of their functionality. Of course, here are presented only the objects effectively related to the SAADI system. A series of objects, created in order to extend the objects hierarchy of the Borland Delphi system, are not displayed here.

Object	Description
+--TVector	Allocation of a vector in Heap
| \--TMatrix	Allocation of a matrix in Heap
+--TGrafic	Projection of 2-D data on text screen
| |--TKarhunen	The same, s-D data, Karhunen-Loewe
| |--TSammon	The same, s-D data, Sammon
| \--TRegr	Computes essential regression parameters
+--TGGrafic	Projection of 2-D data on graphic screen
| \--TGKarhunen	The same, s-D data, Karhunen-Loewe
+--TReduc	Generic object for dimensionality reduction
| |--TRedKar	Dimension reduction, Karhunen-Loewe
| |--TRedOrd	Dimension reduction, importance coef.
| \--TNormal	Building of normalized data
+--TFuzzyPct	Unsupervised generalized classifier, points

| |--TFuzzyLin	The same, convex combination prototypes
| |--TFuzzyAda	The same, adaptive prototypes
| |--TFuzzyElp	The same, ellipsoidal prototypes
| \--TFuzzyMul	The same, classical partition prototypes
+--TClasPct	Unsupervised hierarchic classifier, points
| |--TClasLin	The same, convex combination prototypes
| |--TClasAda	The same, adaptive prototypes
| |--TClasElp	The same, ellipsoidal prototypes
| |--TClasMul	The same, classical partition prototypes
| |--TIsoPct	Unsupervised horizontal classifier, points
| | |--TIsoLin	The same, convex combination prototypes
| | |--TIsoAda	The same, adaptive prototypes
| | |--TIsoElp	The same, ellipsoidal prototypes
| | |--TIsoMul	The same, classical partition prototypes
| | \--TRegPct	Fuzzification component, point prototypes
| | |--TRegLin	The same, convex combination prototypes
| | |--TRegAda	The same, adaptive prototypes
| | |--TRegElp	The same, ellipsoidal prototypes
| | \--TRegMul	The same, classical partition prototypes
| \--TSimPctIA	Cross-classifier, initial algorithm, variant A
| |--TSimPctAA	The same, associative algorithm, variant A
| |--TSimPctIB	The same, initial algorithm, variant B
| | \--TSimPctAB	The same, associative algorithm, variant B
| \--TSimPctIC	The same, initial algorithm, variant C
| \--TSimPctAC	The same, associative algorithm, variant C
+--TTraining	Supervised classifier, Perceptron method
| |--TKellerHunt	The same, Keller-Hunt method
| |--TRelaxFuzzy	The same, Relaxation method
| |--TRelaxFuzzyVar	The same, variation of Relaxation
| |--TTrGallant	The same, Gallant method
| |--TWidrowHoff	The same, Widrow-Hoff method
| |--THoKashyap	The same, Ho-Kashyapp method

|--THoKashyapVar	The same, variation of Ho-Kashyapp
|--TTestTraining	The same, obiect for testing new methods
\--TFuzTrain	Fuzzy decision supervised classifier, restricted
|--TFuzKNN	The same, nearest k neighbours
\--TFuzNProt	The same, nearest prototype

Table 1: The objects hierarchy of the SAADI system

14. Conclusions

This system presents under a unitary conception different aspects of the classification theory: the projection in bidimensional space, in order to facilitate the visual inspection of data; the selection of relevant characteristics; the projection of data into a space having a reduced dimension; the fuzzy unsupervised clustering, both hierarchical and horisontal; the fuzzy regression algorithm; the supervised clustering using both classical and fuzzy sets; fuzzy decision supervised clustering; graphical versions of these classifiers.

Among the most important facilities of the system we mention:

- being programmed in *Borland Delphi 1.0* and using the Object Oriented Programming and Windows Programming, the system allows to be extended with minimal programming effort;
- moreover, wherever possible, the system does not contain redundant code, meaning that we based ourselves on the facilities of the Objects Oriented Programming;
- because the whole system is based on the creation of two objects for implementing the notions of vector amn matrix using dynamical memory allocation, the system does not have statical limits in what it concerns the dimensions of input data; these depend only on the availability of the Heap;
- the simple structure of the data files and results files; thus it is possible to pipe different components of the system, i.e. to have the output of one component as the input for another;

- the possibility to obtain a file with the history of all the operations performed by each component of the system and not only the final result;
- the independence of the system components from one another, as well as their independence with respect to the moment of setting their working characteristics; thus, it is possible to run different classifications using the same working characteristics or with minimal value changes; it is also possible to reiterate, omit or execute them in any order, with the single condition for this order to be logical;
- the availability of the graphical versions of all the implemented algorithms; thus, it is possible to study the working evolution of the algorithms, and this is very interesting both scientifically and didactically speaking.

This system has been successfully used in the research activity, as it follows:

- for optimally selecting the solvents systems [10];
- for studying the Roman pottery (terra sigillata) [6];
- for classifying different Greek muds [1, 12];
- for studying the importance of fuzzy regression in chemistry [7];
- for studying the Mendeleev's periodic system elements and for generating a fuzzy system of elements [9, 11, 2].

Another series of applications of the fuzzy clustering theory is in study. For these applications, we are also using the capacities offered by the system.

References

[1] DUMITRESCU, D., POP, H. F., AND SÂRBU, C. Fuzzy hierarchical cross-classification of Greek muds. *Journal of Chemical Information and Computer Sciences 35* (1995), 851–857.

[2] HOROWITZ, O., POP, H. F., AND SÂRBU, C. Pattern recognition of chemical elements. *Journal of Chemical Information and Computer Sciences* (1996). To appear.

[3] POP, H. F. *Intelligent Systems in Classification Problems.* PhD thesis, "Babes-Bolyai" University, Faculty of Mathematics and Computer Science, Cluj-Napoca, 1995.

[4] POP, H. F. Supervised fuzzy classifiers. *Studia Universitatis Babeş-Bolyai, Series Mathematica 40,* 3 (1995), 89–100.

[5] POP, H. F. A new class of fuzzy algorithms: Fuzzy 1-Means. *Mathware and Soft Computing (Universitat Politecnica de Catalunya),* (1996). To appear.

[6] POP, H. F., DUMITRESCU, D., AND SÂRBU, C. A study of Roman pottery (terra sigillata) using hierarchical fuzzy clustering. *Analitica Chimica Acta 310* (1995), 269–279.

[7] POP, H. F., AND SÂRBU, C. A new fuzzy regression algorithm. *Journal of Analytical Chemistry 68* (1996), 771–778.

[8] POP, H. F., AND SÂRBU, C. The fuzzy hierarchical cross-clustering algorithm. Improvements and comparative study. *Journal of Chemical Information and Computer Sciences* (1996). To appear.

[9] POP, H. F., SÂRBU, C., HOROWITZ, O., AND DUMITRESCU, D. A fuzzy classification of the chemical elements. *Journal of Chemical Information and Computer Sciences 36* (1996), 465–482.

[10] SÂRBU, C., DUMITRESCU, D., AND POP, H. F. Selecting and optimally combining the systems of solvents in the thin film cromatography using the fuzzy sets theory. *Revista de Chimie 44*, 5 (1993), 450–459.

[11] SÂRBU, C., HOROWITZ, O., AND POP, H. F. A fuzzy cross-classification of the chemical elements, based both on their physical, chemical and structural features. *Journal of Chemical Information and Computer Sciences 36* (1996), 1098–1108.

[12] SÂRBU, C., AND POP, H. F. Fuzzy classification of Greek muds. *The Analyst* (1996). Submitted.

"BABEŞ-BOLYAI" UNIVERSITY, FACULTY OF MATHEMATICS AND COMPUTER SCIENCE, RO-3400 CLUJ-NAPOCA, ROMANIA

E-mail address: hfpop@cs.ubbcluj.ro

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

ADJUNCTIONS AND DATA COLLECTIONS

R. TRÎMBIŢAŞ

Dedicated to Professor Ştefan I. Niţchi

Abstract. We introduce data collections through adjoints. Weak, zero-element and strong data collection types and aggregation operators are introduced. Ones prove that adjoints lead to data collections. Their properies are studied and we prove some algebraic properties useful to database query optimization. Thus we find again classical colection types: lists, trees, sets.

1. Introduction

Data collections play a central role in Database Theory. A database programming language (DBPL) can model application domains most naturally if it has several collection (bulk) types, e.g., lists, sets trees and so on. The first demonstration that a collection type could be accommodated satisfactorily in a strongly typed programming language was Pascal-R [15]. The collection type concept supports two essential activities. First, it allows regularity in structure to be described. Second, it supports powerful and succinct notations for computing with such regular structures. The requirements for collection data types are summarized and argued about in [2]. A type system for collections tends to be rich and complex, since constructs must be provided to declare, construct, inspect and update instances of each collection type. One needs to control the complexity of such type systems by exploiting operations and properties common to a variety of collection types. We think Category Theory is a possible background to describe regularities, developing algebras for collection types and improving efficiency.

This paper tries to introduce definitions for data collections and study and model them using adjoints. We shall show how data collections can be modelled through adjunctions and how adjunctions lead to collection operators whose algebraic properties are studied. These properties may be used for query optimization.

Received by the editors: October 20, 1996.

1991 *Mathematics Subject Classification.* 18A40, 68P15.

1991 *CR Categories and Descriptors.* H.2.1 [Database Management]: Logical Design - data models; H.2.3 [Database Management]: Languages - data manipulation languages, query languages.

2. Basic Notions

In [4] data collections are defined as follows:

Definition 2.1. *A* **data collection** *is a parametrized abstract data type (with type parameter T) of type C(T) with the following operations:*

- *A family of data constructors* $C^n : T^n \to C(T)$ *with the arity* n, *for each* $n \geq 0$.
- *Functions over data collections of type C(T).*
- *Observers –*
  - *(a) selectors: functions from data collections of type C(T) to type T;*
  - *(b) predicates over C(T).*

The algebra of the data collections, that is the semantics of the data constructors and other operations, is defined by a set of first-order Horn clause axioms over the data constructors, functions and observers.

We shall use other definitions for data collections, and our approach will be categorical. For notions of Category Theory the reader may consult [7, 8, 14]. Also the paper [12] illustrates how Category Theory notions and constructions can be systematically used in Computer Science.

Definition 2.2. *A* **weak data collection** C *is a parametrized abstract data type (with type parameter T)*

$$C(T) = < \tau, [x], ++, \text{aggr} >$$

where

- τ *is the set of data collections over T;*
- $[x]$ *is the singleton (single element) collection;*
- $++ : \tau \times \tau \to \tau$ *is the concatenation operator of two collections;*
- aggr *is the aggregation operator defined as follows: if* $f: T \to S$ *and* $\oplus : S \times S \to S$ *is a binary operation, then* $\text{aggr}(f, \oplus) : \tau \to S$ *(i.e.* $\text{aggr} : (T \to S) \times \tau \to S$*) has the following properties:*

$$\text{aggr}(f, \oplus)([x]) = f(x) \tag{1}$$

$$\text{aggr}(f, \oplus)(x{+}{+}y) = \text{aggr}(f, \oplus)(x) \oplus aggr(f, \oplus)(y), \quad x, y \in \tau. \tag{2}$$

Definition 2.3. *A* **zero-element data collection** *C is a parametrized abstract data type (with type parameter T)*

$$C(T) = < \tau, [], [x], ++, \text{aggr} >$$

where

- $< \tau, [x], ++, \text{aggr} >$ *is a weak data collection;*
- $[]$ *is the empty collection and*

$$\forall x \in \tau \qquad x{+}{+}[] = []{+}{+}x = x. \tag{3}$$

- aggr *is the aggregation operator defined as follows: if $f : T \to S$ and $\oplus : S \times S \to S$ is a binary operation with neutral element u, then* $\text{aggr}(f, \oplus) : \tau \to S$ *(i.e.* $\text{aggr} : (T \to S) \times \tau \to S$ *) which verifies (1), (2) and*

$$\text{aggr}(f, \oplus)([]) = u. \tag{4}$$

Definition 2.4. *A* **strong data collection** *is a zero-element data collection such that the concatenation is associative, that is*

$$\forall x, y, z \in \tau \qquad (x{+}{+}y){+}{+}z = x{+}{+}(y{+}{+}z). \tag{5}$$

The definition of strong data collection requires $< \tau, ++, [] >$ be a monoid. A good model for strong collections is the free monoid over T (directly applicable to lists). The notion of strong collection is similar to monoid collection defined in [9, 10, 11], but there the approach is not categorical.

Remark 2.5. Each strong data collection is also a zero-element data collection; each zero-element data collection is also a weak data collection.

In the sequel $[x_1, \ldots, x_n]$ will denote the finite collection having the elements $x_1, \ldots x_n$. Some data collections properties from [4] are expressed using concatenation:

- **permutability** – a data collection type is permutable if
  x ++y = y ++x
- **duplicate elimination** – a data collection type eliminates duplication if ++is idempotent, that is
  x ++x = x

- **null value elimination** – a zero-element data collection type eliminates null values if

  x ++ []= x.

Remark 2.6. Each zero-element data collection elimines null-values.

3. Data collections through adjunctions

Let $\mathcal{T}$ be the category of types (whose morphisms are usual functions), $\mathcal{A}$ an arbitrary category and $< F, U, \eta, \varepsilon >$ an adjunction where $U : \mathcal{A} \to \mathcal{T}$, $F : \mathcal{T} \to \mathcal{A}$ are functors such that $F \dashv U$, and η and ε are the unit and respectively the counit of the adjunction. Let us suppose $\mathcal{A}$ is a category of sets with structure having at least one binary operation. Let $U : \mathcal{A} \to \mathcal{T}$ be the forgetful functor. It has a left adjoint $F : \mathcal{T} \to \mathcal{A}$. If $T \in \mathrm{Ob}\mathcal{T}$ then F(T) is the free-algebra generated by T.

Theorem 3.1. *If $U : \mathcal{A} \to \mathcal{T}$ is the forgetful functor and F is his left adjoint, then the adjunction $< F, U, \eta, \varepsilon >$ determines a weak data collection.*

Proof. If $F \dashv U$ then there exist $\eta : id_{\mathcal{T}} \to UF$ (the unit of the adjunction) such that $\forall X \in \mathrm{Ob}\mathcal{T}$, $\forall Y \in \mathrm{Ob}\mathcal{T}$, $\forall f : X \to UY$ $\exists! f^{\#} \in Hom_{\mathcal{A}}(F(X), Y)$ such that the following diagram commutes

$$\begin{array}{ccc} X & \xrightarrow{\eta_X} & UF(X) \\ & \searrow^{f} & \downarrow U(f^{\#}) \\ & & U(Y) \end{array} \qquad (6)$$

(see [8, 14, 13]).

For each $T \in Ob\mathcal{T}$, $F(T)$ is the free-algebra with one binary operation generated by T. We shall take $\tau = UF(T)$, and the concatenation will be the binary operation over $F(T)$. The singleton collection will be given by the unit of adjunction, that is $\forall x \in T$ $[x] = \eta_T(x)$.

The aggregation is defined as follows: for $f : X \to U(Y)$ we have

$$\mathrm{aggr}(f, \oplus) = f^{\#}.$$

Since $f^{\#} \in Hom\mathcal{A}$ we obtain

$$f^{\#}(x{+}{+}y) \;=\; f^{\#}(x) \oplus f^{\#}(y)$$

that is (2).

The commutativity of diagram (6) implies for $x \in X$

$$(U(f^{\#}) \circ \eta_X)(x) = U(f^{\#})([x]) = f^{\#}([x]) = f(x)$$

that is (1).

Example 3.2. If $\mathcal{A}$ is the category of sets with one binary operation (subject to no restrictions) and the arrows are such binary algebra homomorphisms we obtain binary labelled trees.

An ordered binary rooted tree (OBRT) is a binary rooted tree which has an additional linear order structure (referred to as left/right) on each set of siblings. An X-labelled OBRT (LOBRT/X) is an OBRT together with a function from the set of terminal nodes to X. A free algebra $F(X)$ expression has the usual representation as a binary tree. For example $(xy)z$ is represented in figure 1.

The concatenation of two trees t_1 and t_2 (or of two expressions) is the binary tree

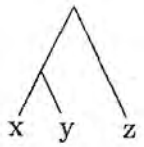

FIGURE 1. Expression

having t_1 as left subtree and t_2 as right subtree. □

If $\mathcal{A}$ is a subcategory of the category **Mon** of monoids we get an analogous of Theorem 1 for strong collections.

Theorem 3.3. *If $\mathcal{A}$ is a subcategory of* **Mon**, *$U : \mathcal{A} \to \mathcal{T}$ is the forgetful functor and F is his left adjoint, then the adjunction $< F, U, \eta, \varepsilon >$ determines a strong data collection.*

Proof. It is a variant of theorem 3.1 proof. The unique morphism $f^{\#}$ from diagram (6) is a monoid-homomorphism and for $T \in Ob\mathcal{T}$, $F(T)$ is the free monoid over T. Since $(F(T), ++, [])$ is a monoid we obtain (3) and (5); the commutativity of (6) implies (2), and $f^{\#}$ monoid-homomorphism implies (1) and (4).

Theorem 3.4. *If* **Bu** *is the category of algebras with one binary operation and neutral element, $U :$ **Bu** $\to \mathcal{T}$ is the forgetful functor and F is his left adjoint, then the adjunction $< F, U, \eta, \varepsilon >$ determines a zero-element data collection.*

The proof is analogous to that of theorem 3.3.

Example 3.5. Let add to collections from example 3.2 the empty tree (having no node). If z is the empty tree, the concatenation is extended as follows:

$$z{+\!+}t \;=\; t{+\!+}z \;=\; t$$

for each tree t.

Let $(B, \oplus, u)$ be an algebra with a binary operation $\oplus$, neutral element u and $f : A \to B$. The aggregation is extended as follows

$$\mathrm{aggr}(f, \oplus)(z) \;=\; u.$$

4. Algebraic properties of aggregation

Theorem 4.1. *If $f : X \to U(Y)$ then the following identity holds*

$$\mathrm{aggr}(f, \oplus) \;=\; \varepsilon_Y \circ F(f) \tag{7}$$

Proof. Applying F to (6) we have

$$\begin{array}{ccc} F(X) & \xrightarrow{F(\eta_X)} & FUF(X) \\ & \searrow^{F(f)} & \downarrow{\scriptstyle FU(f^\#)} \\ & & FU(Y) \end{array} \tag{8}$$

The naturality of ε gives us

$$\begin{array}{ccc} FUF(X) & \xrightarrow{FU(f^\#)} & FU(Y) \\ \downarrow{\scriptstyle \varepsilon_{F(X)}} & & \downarrow{\scriptstyle \varepsilon_Y} \\ F(X) & \xrightarrow[f^\#]{} & Y \end{array} \tag{9}$$

that is,

$$\begin{aligned} \varepsilon_Y \circ F(f) &= \varepsilon_Y \circ FU(f^\#) \circ F(\eta_X) \qquad (\text{from}(8)) \\ &= f^\# \circ \varepsilon_{F(X)} \circ F(\eta_X) \qquad (\text{from}(9)) \\ &= f^\# \qquad (\epsilon_F \circ F_\eta = id)\ \square \end{aligned}$$

Remark 4.2. Theorem 4.1 also holds for strong collections.

Proposition 4.3. The following holds

$$\text{aggr}(\eta, ++) = id_{F(.)} \tag{10}$$

Proof. From theorem 4.1 and adjunction we have

$$\text{aggr}(\eta, ++) = \varepsilon_{F(T)} \circ F(\eta_T) = id_{F(T)}. \square$$

Remark 4.4. The identity (10) also holds for strong collections.

Proposition 4.5. *If $h \in Hom_A(T, S)$ where $T = (T, \oplus)$ and $S = (S, \otimes)$ are algebras with one binary operation, then for each $f : A \to U(T)$ we have*

$$h \circ aggr(f, \oplus) = aggr(h \circ f, \oplus) \tag{11}$$

Proof. From theorem 4.1 we have

$$\begin{aligned} \text{aggr}(f, \oplus) &= \varepsilon_T \circ F(f) \\ \text{aggr}(Uh \circ f, \otimes) &= \varepsilon_S \circ F(Uh \circ f) \end{aligned}$$

and

$$\begin{aligned} h \circ \varepsilon_T \circ F(f) &= \varepsilon_S \circ FU(h) \circ F(f) \qquad (\text{ naturality of } \varepsilon) \\ &= \varepsilon_S \circ F(U(h) \circ f) \qquad (F \text{ functor}) \\ &= \text{aggr}(Uh \circ f, \otimes) \\ &= \text{aggr}(Uh \circ f, \otimes), \end{aligned} \tag{12}$$

since $Uh = h$. □.

Remark 4.6. Formula 11 also holds for zero-element and strong data collections.

Proposition 4.7. If $g : A \to B$ and $f : B \to C$ where $(C, \oplus)$ is an algebra with one binary operation then

$$\text{aggr}(f, \oplus) \circ F(g) = \text{aggr}(f \circ g, \oplus) \tag{13}$$

Proof. From theorem 4.1, because F is a functor we have

$$\begin{aligned} \text{aggr}(f, \oplus) \circ F(g) &= \varepsilon_C \circ F(f) \circ F(g) \\ &= \varepsilon_C \circ F(f \circ g) \\ &= \text{aggr}(f \circ g, \oplus). \square \end{aligned}$$

5. The importance of aggregation

The aggregation operator first appeard in John Backus' article [3]. In [4] is called *pump*; in [5] *aggr*. In [18] appears as *fold* and is the basic operator for database query and optimization. Its importance is that aggregation axioms and identities (7), (10), (11), (13) can be used for query optimization purpose. Also, the usual operators of relational algebra ([17, 16, 1]) can be expressed by aggregation as follows.

P1. Collapse. If we have a data collection type

$$C = (\tau', [], [x], ++, \text{aggr})$$

where τ' is a collection of τ-type collections then the collapsing can be expressed as

$$\text{collapse}(x) = \text{aggr}(id, ++)(x).$$

P2. Selection. If p is a predicate, we have for strong collections

$$\sigma_p(x) = \text{aggr}(f_p, ++)$$

where $f_p = \lambda x.\text{if } p(x) \text{ then } [x] \text{ else } []$.

P3. Map or apply-to-all.

$$\text{map}(f, x) = \text{aggr}(f, ++)(x).$$

Remark 5.1. Map is intimately related to the free functor and adjunction. In fact we may write

$$\text{map}(f, .) = F(f).$$

P4. Project.

$$\text{project}_{a_1,\ldots,a_n}(x) = \text{map}(\lambda\{a_1 = x_1, \ldots, a_n = x_n, \ldots\}.$$
$$\{a_1 = x_1, \ldots, a_n = x_n\}, x).$$

P5. Join. We introduce

$$\text{filtermap}(p, f, x) = \sigma_p(\text{map}(f, x))$$

We have

$$\text{join}(xs, ys, f, g, h) = \text{map}(\psi, xs)$$

where

$$\psi = \lambda x.\text{filtermap}((\lambda y.f(x) = g(y)), h(x,y), ys).$$

P6. Powerset. If $C(T)$ is the set-type collection over T, then the set of all sets over T (T finite) can be expressed by:

$$\begin{aligned}
&\text{power} : C(T) \to C(C(T)) \\
&\text{power}([\,]) = [[]] \\
&\text{power}\big(\text{add}(y, X)\big) = \text{power}(X) {+\!+} \text{map}(h)\big(\text{power}(X)\big)
\end{aligned}$$

where $h(c) := \text{add}(y, C)$ and $\text{add}(y, C) = [y] {+\!+} C$, $y \in T$, $c \in C(T)$.
The generalization of this construction for other collection types is easy.

Other operations. The following operations may be expressed through aggregation:

- set membership

$$e \in X = \text{aggr}(f, \vee)(x) \qquad \text{where } f = \lambda z.(z \doteq e);$$

- set difference

$$x \backslash y = \text{aggr}(f, \cup)(x) \qquad \text{where } f = \lambda z.\text{if } \neg(z \in y) \text{ then } [z] \text{ else } [\,];$$

- set intersection

$$x \backslash y = \text{aggr}(f, \cup)(x) \qquad \text{where } f = \lambda z.\text{if } (z \in y) \text{ then } [z] \text{ else } [\,];$$

- quantifiers

$$\begin{aligned}
(\forall x.p(x))(y) &= \text{aggr}(f, \wedge)(y) \\
(\exists x.p(x))(y) &= \text{aggr}(f, \vee)(y) \qquad \text{where } f = \lambda z.p(z).
\end{aligned}$$

6. Applications

We can find again classical data collection types taking various target categories for F functor.

Example 6.1. Let $\mathcal{A}$ be the categories of algebras with one binary operation. We obtain the data collection type from example 3.2. □

The next examples are on strong data collections.

Example 6.2. If $\mathcal{A}$ is the category of algebras with one binary operation having neutral element we obtain the data collection type from example 3.5. □

Example 6.3. If $\mathcal{A} = \mathbf{Mon}$, where **Mon** is the category of monoids we obtain list data collection types.

If $M \in Ob\mathcal{T}$, then $F(M)$ is the free-monoid generated by M (the set of lists having M-type elements), and for $f \in Hom\mathcal{T}$, $F(f)$ applies f to each element of the list (*apply-to-all* or LISP *mapcar* function). In fact F is the free-functor, and his right adjoint U is the forgetful functor.

If x is the list $[x_1, \ldots, x_n]$ and $f : X \to Y$, where $(Y, \oplus)$ is a monoid, then

$$\mathrm{aggr}(f, \oplus)(x) = f(x_1) \oplus \ldots \oplus f(x_n).$$

The concatenation is the usual list concatenation operation, the empty collection is the empty list, and the single-element collection is the one-element list. □

Example 6.4. If $\mathcal{A} = \mathbf{CMon}$, where **CMon** is the category of commutative monoids, we obtain the multiset(bag) data collection types. The empty collection is the empty multiset, and the concatenation is the multiset union (that is, the number of occurrences of an element is the number of occurrences in the first multiset plus the number of occurrences in the second).

If x is the multiset $[x_1, \ldots, x_n]$ with X-type elements and $f : X \to Y$, where $(Y, \oplus)$ is a commutative monoid, then

$$\mathrm{aggr}(f, \oplus)(x) = f(x_1) \oplus \ldots \oplus f(x_n). \square$$

Example 6.5. If $\mathcal{A}$ is the category **CUSL** of upper complete semilattices we obtain set data collection types. The empty collection is the empty set, the singleton collections are singleton sets, and the concatenation is the usual union.

If $\{x_1, \ldots, x_n\}$ is a set with X-type elements, $f : X \to Y$, where $(Y, \vee)$ is an upper complete semilattice then

$$\mathrm{aggr}(f, \vee)(x) = f(x_1) \vee \ldots \vee f(x_n) \square$$

Example 6.6. If $\mathcal{A}$ is the category of monoids with one binary idempotent operation, then we obtain *oset* type data collections (lists without duplicates). The empty and singleton collection are the same as for ordinary lists, but the definition of concatenation is $x{+}{+}y = x \cup (y - x)$ where $y - x$ is the list of elements in y, but not in x. □

Example 6.7. If $\mathcal{A}$ is the category of ordered idempotent monoids we obtain sorted collections. The concatenation is list merging. A variant of these collection type parametrized by a function f whose range has a partial order "$\leq$" given by $x \leq y \Longleftrightarrow f(x) \leq f(y)$, and called *sorted[f]* appear in [10] and [11]. □

Remark 6.8. Some classical aggregation operations for databases, such as count or sum are exception from the frame of example 6.2. They could be defined as $\text{count}(x) = \text{aggr}(f, +)$, where $f(y) = 1$, and $\text{sum}(x) = \text{aggr}(id, +)$.

We have

$$\text{count}(\{2\}) = 1 = \text{count}(\{2\} \cup \{2\}) \neq \text{count}(\{2\}) + \textit{count}(\{2\}) = 2$$

$$\text{sum}(\{2\}) = 2 = \text{sum}(\{2\} \cup \{2\}) \neq \text{sum}(\{2\}) + \text{sum}(\{2\}) = 4$$

For these cases f is not a homomorphism of complete upper semilattices. □

There is another way to define aggregation operators, having theorem 4.1 as starting point. If $f : T \to S, \oplus : S \times S \to S$ is a binary operation and $[x_1, \ldots, x_n] \in F(T) = \tau$ we define

$$\varepsilon_S([x_1, \ldots, x_n]) = x_1 \oplus \ldots \oplus x_n.$$

The definition of aggregation operator is now

$$\text{aggr}(f, \oplus) : \tau \to S, \quad \text{aggr}(f, \oplus) = \varepsilon_S \circ F(f)$$

We shall call this operator generalized aggregation operator. In fact, this is the classical aggregation operator as defined in Functional Programming. This operator will have the property (2) or (4) only if $f : T \to S$ is a homomorphism of appropriate algebras.

Alternatively, the aggregation operator can be seen as a mapping from a colection type $< \tau, [x], ++, \text{aggr} >$ to a binary operation algebra $(S, \oplus)$. It will have the desired properties if the category which has the algebra $(S, \oplus)$ as object is a subcategory of the category corresponding to the collection type. This property is called *well-definedness* and it is studied in [9, 10, 11, 6], but the way of expression is not categorical. For sum and count, $(\mathbb{N}, +)$ is not an upper complete semilattice, but a commutative monoid. Thus sum and count will be well-defined for lists and multisets and ill-defined for sets. Adjunction leads to well-definedness.

References

[1] S. Abiteboul, R. Hull, V. Vianu – *Foundations of Databases*, Draft book, 1994.

[2] M. P. Atkinson, P. W. Trinder, D. A. Watt - Bulk Type Constructor, *Technical Report*, FIDE/93/61, 1993.

[3] John Backus – Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *CACM* 21(8), pp. 613-641, 1978.

[4] Catriel Beeri, Yoram Kornatzky – Algebraic Optimization of Object-Oriented Query Languages, *Report* 91-6, Hebrew University of Jerusalem, Israel, 1991.

[5] Catriel Beeri, Tova Milo – Functional and predicative programming in OODB's, *PODS 92*, 1992.

[6] P. Buneman, S. Naqvi, V. Tannen, L. Wong – Principles of programming with complex objects and collectin types, *TCS* no. 1, pp. 1-46, 1995.

[7] Michael Barr, Charles Wells – *Toposes, Triples and Theories*, Springer Verlag, Berlin, Heildelberg, Tokyo, 1985.

[8] Michael Barr, Charles Wells – *Category Theory for Computing Science*, Prentice-Hall, 1990.

[9] L. Fegaras – A Uniform Calculus for Collection Types, *Technical Report* No. CS/E 94-030, OGI, December, 1994.

[10] L. Fegaras, D. Maier – Towards an Effective Calculus for Object-Oriented Query Languages, *ACM SIGMOD International Conference on Management of Data*, San-Jose, May, 1995.

[11] L. Fegaras, D. Maier – An Algebraic Framework for Physical OODB Design, *5th International Workshop on Database Programming Languages*, Gubbio, Italy, 1995.

[12] Joseph A. Goguen – A categorical manifesto, *Math. Struct. in Comp. Science*, vol. 1, pp. 49-67, 1991.

[13] Saunders MacLane – *Categories for the Working Mathematicians*, Springer-Verlag, 1971.

[14] Benjamin C. Pierce – A Taste of Category Theory for Computer Scientist, *Research Report* CMU-CS-88-203, Carnegie-Mellon University, Pittsburgh, 1988.

[15] J. W. Schmidt – Some high level language constructs for data of type relation. *ACM Transactions on Database Systems* 2(3), pp. 247-261, 1977.

[16] J. D. Ullman – *Principles of Database Systems*, Computer Science Press, 1982.

[17] J. D. Ullman – *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

[18] Bennet Vance – Towards an Object-Oriented Query Algebra, *Technical Report* CS/E 91-008, Oregon Graduate Institute, 1991.

"BABEŞ-BOLYAI" UNIVERSITY, FACULTY OF MATHEMATICS AND COMPUTER SCIENCE, RO-3400 CLUJ-NAPOCA, ROMANIA

E-mail address: tradu@cs.ubbcluj.ro

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

SOFTWARE QUALITY MANAGEMENT: TO UNDERSTAND AND TO INTRODUCE (II)

E. BALLA

Abstract. Following a brief exposition of the history of software-quality definitions, today's most popular approach to the problem was presented in Part I: building quality management systems according to ISO 9000 Series. Some advantages and disadvantages of this trend were shown. In this part, using results of software quality oriented research and practical training done by the author in Hungary, some problems and their possible solutions are sketched to start introducing software quality management. The problem are similar in most Eastern European countries, therefore we presume that presented solutions could be applicable as well.

1. Software quality management

Software quality management is described not only in ISO 9000-3 [1], but in many other standards too: IEEE, EN and some more Military Standards.

Software quality managers seem therefore to have an easy job developing and maintaining quality management systems: theoretically they only have to read the many standards and apply them. But things are far from being that simple. Standards provide prescriptions for quality system elements and quality control elements – in general terms –, but they don't give the sequence of the activities to be done, in fact they let the quality manager guess not only how to do things, but also what exactly to do in order to develop a quality system that fits ISO requirements. If the work is done by experienced people, the prescriptions of standards are probably of real help, but unfortunately this isn't the typical situation.

In the following we present some problems in developing and introducing software quality management systems and try to sketch some solutions for solving them. We

Received by the editors: December 15, 1994.

1991 *Mathematics Subject Classification.* 68M15, 68M20, 68N99.

1991 *CR Categories and Descriptors.* D.2.9 [Software Engeneering]: Management Software – quality assurance; K.6.2 [Management of Computing and Information Systems]: Software Management.

[1] Guidelines for the application of ISO 9001 to the development, supply and maintenance of software

proceed by taking into account the Hungarian situation, which can be considered typical for most Eastern-European countries.

1.1. **Before ISO 9000.** For a better understanding of the present situation, we have to consider the *software-quality oriented research and work done before ISO 9000 appeared.*

We can state that despite the former isolation of Eastern-European countries, the quality-research in these countries has followed the steps of Western development. It's worth to note that the first attempts for identifying software quality attributes and their relationships were made in 1977 in Hungary – that means 7-9 years of delay in comparison with the USA. Western-European situation has also to be taken into account: SIEMENS, for instance proceeded only in 1975-76 in identifying these elements, that places Hungary at two years of delay. In 1977 in Hungary Basic Quality Requirements for software have been stated, and by 1985 some quality-measuring programs also have been developed. These attempts fit in the world-trend of measuring software quality through some well-identified attributes and in application of the product-based definition of software quality.

1.2. **What's happening now – and what can be done.** Like other companies in business life, Eastern European software companies join the trend of developing quality management systems according to ISO 9000.

1.2.1. *Quality organizations in Eastern Europe.* In the Eastern European countries national quality organizations have been organized, which have insured the translation of ISO 9000 standards, are organizing conferences, meetings on quality-related themes [2]. In most cases there are foreign registering organizations working in these countries [3]. National quality organizations and research institutes also get more and more envolved in enhancing application of ISO 9000 under the spacial conditions given in each country. The *Processus* project, for instance, founded in Slovenia, supported by 11 Slovenian software companies, has the scope of developing a tool for auditing and goal definition. (The project and the tool are presented in [Gyorkos94]. *Tipically no software company has been registered yet.*

[2] In Hungary, for instance, in 1993 there have been about 8 quality conferences – the largest being the annual Hungarian Quality Week – each of them involving the participation of about 80-100 companies

[3] In Hungary: Bureau Veritas, TÜV Rheiland, TÜV Bayern, TÜV Hannover, SGS (foreign), and a Hungarian registering organization (Mertcontrol)

1.2.2. *Teaching quality management.* As we briefly presented above, ISO 9000 operates with many quality-related terms, which have to be *well understood* and *correctly applied*, as in other fields of working and academic life. According to [Geiger93] in the field of quality management this "necessity is of even higher importance because of the lack of basic knowledge from education".

In many Eastern European countries quality management is taught only on off-school courses, which, for the moment, are too expensive to afford for individuals and for small companies (a 2-day course costs about 650$). Consultancy alone for developing a software quality management system costs about 30.000$, therefore such systems are developed mainly by "self made" quality managers, who, in best case, have joined some quality-oriented conferences and read the available theoretical material. For these people *standard's definitions are often not relevant.*

At some universities (Technical University Budapest – Hungary –, Technical University Maribor - Slovenia - for instance) basic project management and software development methods are being taught. *Restructuring the quality-related educational system* would be a first step to change the present situation. Postgraduate courses have to be replaced by usual educational elements, first in universities, than in high schools too. Some models are available in western countries[4].

2. Developing software quality management systems

The first step in developing a (software) quality management system is **identifying the present situation**, in order to sketch what is to be done.

We try to describe the present situation and discuss some elements that can prepare introduction of software quality management according to 4 problem-groups, which, in our opinion, are basic when a software quality management system is to be developed.

These groups are:

a. *Applying a structured methodology for system design and development*

b. *Applying a project management methodology*

c. *Organizing other (non-project-related) activities at the company*

i. *services*

external (e.g.,courses, consultancy)

[4]Zulema Lopes Pereira [Pereira93] presenting the Portugaal quality-situation mentioned that Portugal's national quality policy hass been stated in 19911, and in 1993, every university student had weekly at least 3 hours of quality management courses

internal (e.g.,network maintenance)

ii. *information system*

iii. *human relationship*

d. *Writing down aspects regarding the elements a.–c. in a clearly stated and easy-to-understand manner*

2.1. **Project-related activities.** In the following we present some problems connected with the first two elements (a.-b.) of the above described group.

The *structured system developement and project management methodologies* started to develop in the eighties. These methods and methodologies decompose the system development activities – structure them – into stages, phases, steps, define the sequence of the activities. They prescribe precise inputs and outputs for each step. They recommend a top-down approach and a step-by step refinement of the system. They make a strict difference between logical and physical planning. The techniques used are similar. (Entity – relationship modeling, function hierarchy modeling, entity – life – history modeling, data-flow modeling etc.). Nowadays they cannot be used without complex computer-aided tools, which help to transform the logical model into physical model and even into program code[5].

2.1.1. *Structured system development methodologies.* SSADM (Structured Systems Analysis and Design Method) has been developed in the UK by LBMS (Learmonth and Burchett Management Systems), by request of CCTA Central Computer and Telecommunications Agency for a self-testing, teachable methodology, which would use tested system development methodology in the UK[6].

Oracle Case Method* was defined in the early eighties by Richard Barker, the first book about the method was printed in 1989. It has been derived from many years' collective experience delivering systems to clients and building software products.

The two mentioned methodologies differ in the life-cycle-steps covered, in their most recommended techniques, in their notations.

[5]More and more structured system development and project management methodologies become available in Eastern European countries. Presentations in sections 2.1.1 and 2.1.2 does not suggest any hierarchy among these, it just deals with metodologies considered by the author being the most used.

[6]SSADM has been translated into Hungarian, and it is the metodology most probably becoming recommended in Hungary

Oracle CASE* Method	SSADM
Strategy	*Feasibility study*
Analysis	Problem definition
Design*	Project identification
Build	*System analysis*
User Documentation	Analysis of systems operations and curent problems
Transition	Specification of requirements
Production	Selection of technical options
	System design
	Data design
	Process design
	Physical design*

TABLE 1. Project life cycle in two methodologies

Table 1 makes a comparison between the project life-cycle elements covered by the SSADM and Oracle Case* Method.

To be noted that neither SSADM nor Oracle Case* Method cover the entire project life-cycle. A possible model for covering all life-cycle should be worked out by each company, according to the work's characteristics.

2.1.2. *Project management methodologies. PRINCE* (Project Run IN Controlled Environment) is a project management methodology – in fact *the* Project management Standard of the UK government IT departments. The methodology consists of 3 major components that are applied to a project: *organization, planning* and *control.* Using it will lead to a common understanding of responsabilities, of activities, of terms, an efficient and familiar documentation and will reduce dependance on the individiual.

TickIT was prepared by the British Computer Society under contract to the UK Department of Trade and Industry. It isn't exactly a system development nor a project management methodology. It's a guide to software quality management system construction and certification, using ISO 9001. Purchasers' guide, suppliers' guide and auditors' guide provide guidance for quality system elements and quality control elements which can be used not only in developing a software quality management system, but in defining system development and project management steps too.

2.1.3. *Software tools for structured methodologies.* The real help for using such structured methodologies is given by their software tools.

SSADM Engineer is the software tool-set developed by LBMS for supporting SSADM methodology. The current version (SSADM Engineer 5.0, September 1993) runs on Windows 3.1. It operates with software prodeced by Gupta Technologies Inc., using a database for maintaining textual and graphic information about the system to be developed. It supports almost all life cycle functions of SSADM, providing help in transforming logical model into physical model.

Using Oracle Case* Method is supported by the *Oracle CASE* tools, which facilitate the interaction with data maintained in the central database, implemented with the Oracle SQL data base management system. *Case* Dictionary* and *Case* Designer* provide defining and repositoring the data used in logical planning (textual and graphic), while application-generators, such as *Case* Generator for SQL* Forms/SQL* Menu, Report Writer* provide transforming the logical design-related data into running program components.

Project management methodologies are supported by varoius software tools, such as: *Microsoft Project, team Windows, Artemis Prestige.*

Artemis Prestige was developed by Lucas Management Systems. It runs under Windows, co-operating with many types of hardware, using Oracle, Gupta SQL Base database management systems. It provides help for project planning (timing, resources, cost-analysis).

2.1.4. *Using structured system development and project management methodologies.* The first two elements in the problem group presented at the beginning of the chapter (a.,b) seem to be easy to realize. Apparently they would "only" require a management decision regarding the use of a structured system development and a project management methodology. At software-companies the use of computer-aided tools is not a real difficulty.

However, the situation is much more complicated. The majority of analysis and programmers have their "good old" methods to develop a system, and in many cases the customers also have some requirements – differing one from another – which have to be satisfied. So, instead of applying one structured methodology for the whole project, teams rather apply *some elements of certain methodologies*, combining them with others

to satisfy customers' needs. Project managers in most cases are aware of the advanteges of using a single structured methodology, and so is the company management. The main reason that impedes consequent use of a methodology is time – that means money. Companies have to live, and – for instance – SSADM methodology with all its steps, tasks and documents is considered to prolong system-development by such an amount of time, that cannot be afforded by a small company.

It is very important to identify *product life-cycle and system development steps.* While strategy, analysis and development steps are followed in the majority of cases, *project preparing steps* – offer, tender problem definition, first-cut project plan – and *product maintenance steps* – installation, maintenance, analysis of the errors, maintaining user observations, etc. – *are often neglected.* They are not considered to be project elements, therefore time for doing them isn't allocated. Project members have to do these activities "outside" the project, many times parallel to an other project. The rush has negative effects on the success of the project – *the contract is not well-defined, the reasons of success or failure of offers, tenders are not analyzed, user-reactions, observations are not followed and analyzed*, etc. *Marketing plans, cost analysis are seldom done*, and seldom by experts. *Test-plans, configuration* and *change management plans are not being used*, although every project manager feels their lack. Usually *there are few checkpoints in the projects* when the team could discuss its problems with the management.

Users are seldom cooped into reviews – in fact they are only cooped when they insist on it, projects evaluation reviews are seldom organized.

Applying a structured system development and a project management methodology, in my opinion *cannot be achieved at once.* If someone tries to convince system analysts and programmers about the fact that their working methods are bad, they will demonstrate in short time that prescriptions of the quality management system lead to worse results. Nevertheless, there are some elements that we can start with.

Being aware of the lengthening of software product life cycle would be one of these. Project initiation steps and follow-up have been done, the results of tenders and users' observations have to be analyzed and the experience used in forthcoming projects.

Using a structured system development methodology and a project management methodology can also *be introduced step-by-step. SSADM, Oracle Case Method* with their computer-aided tools offer, first of all, an organized working frame. Getting familiar

with their concepts – even without using them – will lead to a change in the mentality of system-developers. They shall notice, for instance, that both models make a strict difference between logical and physical design and prescribe precise inputs and outputs for each step.

Organizing project-related data is possible at any moment. *Recommended directory structure* of all data related to a project can be described, *archiving methods* can be worked out and documented. Employees are to be encouraged to keep, for instance, their project-related incoming mail (electronic and ordinary) separately from the project oriented outgoing mail and from other subject mail.

There are some elements in *project' quality management* that can be applied from the beginning. Project managers, programmers will notice the help given by *configuration management plans, change management plans, test plans* and *test journals, journals of project related activities* or *list of documents used and prepared.* They will find – at first informal – *quality reviews* useful. *Applying corrective actions during a stage, not after finishing it is a basic goal of quality management.*

Project documentation is an important part of project-related activities. The problem is here that documentation in a company is seldom uniform, every team develops its documents according to the needs of the customer. The quality management system has to prescribe the outline and the content of each document type, so, the many types of documents have to be gathered in a uniform system, that fits the needs of every user and every project member.

2.2. **Non-project-related activities.** The problem with the activities sketched in point c. of the problem-group presented at the beginning of this chapter is that they seem to work. Announced courses are held, users' problems are solved sooner or later, system backups are being performed more or less regularly. But, because of poor organization these activities imply much more effort than normal. When a project manager, a system engineer or a secretary goes on holiday, it takes a real effort for someone else to handle his/her job. It's a big problem, when in an office a computer has to be replaced, because no one knows for sure the configuration, the utility-versions that have to be applied.

Such difficulties can be surmounted by appling a well-organized *information system.*

The information system – which is the totality of electronically and manually preserved information – can exist before a quality management system is being developed. In most cases this system is developed together with, or in the best case, it is documented as a part of the quality management system.

Building the information system can start with unifying the documents and documentation used by the company not only in project-related activities, but in any other activity done. Templates should be described not only for project documentation, but for official letters, fax-headers as well.

Employees have to be noticed about every new feature of the "nascent" information system and have to be encouraged to use them and report any connected objections.

Human relationships have an important effect on the company's work. If the working conditions are good, the efficiency of work is higher. Employees have to carry out tasks they are professionally prepared for. Informatics is an even-changing science, so there is no shame in not knowing something. But people are sometimes put to do things are not prepared for, therefore they work much harder, are stressed-they want to prove they fit management's trust–, therefore their working capacity decreases.

Services, other, non-projrct-related activities can be described only in co-operation with people who effectively do the related work. If tehere are some rules, the quality manager should describe them, if there are no rules, he should work them out together with the system engineer, training manager, secretary, etc.

Communication among team members, management relationship with employees has to be organized without being unnecessarily formalized.

Human relationships cannot be changed by the quality management system. However, there is a hope that well-organized activities will diminish stress and extra work – which lead to an improvement of human realtionships at all levels.

2.3. **Writing the quality management system documentation.** At last, but not least, writing the quality management system documentation is an important part of the system. We've mentioned it in point d. of the problem-group presented at the beginning of this chapter because we think it's better describing an existing system than trying to build up the system according to the previously written documentation. Assessment for

registration begins with consulting the documentation, so we could be tempted to write a "perfect" documentation set, according to ISO 9000 prescriptions but without taking into account the existing situation. Such a quality management system cannot be viable.

3. Conclusions

Due to the world-wide acceptance of ISO 9000 Series, we speak more about developing, introducing and maintaining software quality management systems than about the quality of the software product, as defined earlier.

Basic motivation for developing such systems is the wish of the software companies to get registred according to ISO 9000.

With all desire to get registred in the shortest time, I think software quality management systems cannot be applied at once after being developed. Applying every prescription would entirely change a company's activity, causing financial and more crisis.

The best thing that can be done is to involve all employees in the development of the quality system. The management should present the concept as necessary for the company's welfare, the use of which is well motivated. It is recommendable that the motivation should't be ISO 9000 registration, but facilitating employees work.

However, applying the elements described above seems to be possible. These elements are accepted easier probably because of the immediate help they provide both in system development and project management.

References

[Beiczer78] Beiczer, Ö, Szentes, J. Felhasználói programok forráslista alapján történő mininősítése, Softtech sorozat, D26, SzKI, 1978.

[Boehm78] Beohm, B.W., Characteristics of software quality, TRW Series of Software Technology, Vol.1, North Holland, 1978.

[Develin93] Develin & Partners Management Consultants, Freeing the Victims – Achieving Cultural Change Through Total Quality, 1993.

[Feigen93] Feigenbaum, A., EOQ World Conference, Helsinki, in: MMT Tarsasagi Tajekoztato, II. evf. 12. sz. 1993. dec.

[Garvin84] Garvin, D.A., What does "product quality" really mean? Sloan Management review, Fall 1984.

[Geiger93] Geiger, W., Talk Show with ISO 8402? Contemplative thoughts about the new quality vocabulary, EOQ Quality 1/1993, pg. 13-17.

[Genuch91] Genuchten, M. van, Towards a Software Factory. Ph.D. thesis, Eindhoven University of Technology, 1991, ISBN:90 900 4119 2

[Gyorkos94] Györkös, J., A support for auditing the software process, in: Austrian-Hungarian Seminar on Software Engineering Klagenfurt, 7-8 April 1994, pg.7.1.-7.4.

[Havass93] Havass, M., A szoftvervizsgálat múltja és jövője, II. Minőségi Hét, 1993, November 8-10., A konferencia előadásai, II. kötet, pg. 85-101.

[Interl93] The ISO 9000 Guide, Interleaf, 1993.

[McCall78] McCall, J.A., The utility of software metrics in large scale software systems development, IEEE Second software life cycle management workshop, August 1978.

[Oracase] Barker, R, Case* Method, tasks and Deliverables, Addison-Wesley Publishing Company, Revised Edition, 1990.

[Pereira93] Pereira, Z.L., EOQ World Conference, Helsinki, in: MMT Társasági Tájékoztató, II. évf. 12. sz. 1993. dec.

[PRINCE91] Project Run In Controlled Environment, LBMS Delegate Notes, 1991.

[SSADM88] Downs, E., Clare P., Coe I., Structured Systems Analysis and Design Method – Application and Context Prentice Hall International Ltd. 1988.

[Stephens93] Stephens, K., Quality Systems and Certifications – Some Observations and Thoughts, EOQ Quality, 1/1993, pg. 5-12.

[Szentes82] SOMIKA – An automated System for Measuring Software Quality, Proc. of Premier Colloque de Génie Logiciel, AFCET, 261-276 (1982).

[Szentes83] Szentes, J., Qualigraph – A Software Tool for Quality Metrics and Graphic Documentation, Proc. of ESA/ESTEC Software Engineering Seminar, Noordwijk, The Nethderlands, 73-81, 1983.

[Szentes85] Szentes, J., A szoftverminőség és mérése, Számitástechnika-alkalmazási Vállalat, Budapest, 1985.

[TickIT90] Guide to Software Quality Manegement System Construction and Certification using ISO 9001/EN 29001 Part 1 (1978). Issue 1.1 (30 September 1990) British Computer Society UK Department of Trade and Industry.

[ISO9000] Quality management and quality assurance standards – Guide for selection and use, 1987.

[ISO9001] Quality system – Model for quality assurance in design/development, production, installation and servicing, 1987.

[ISO9004] Quality management and quality system elements – Guidelines, 1987.

[ISO9000-3] Quality management and quality assurance standards – Guidelines for the application of ISO 9001 to the development, supply and maintenance of software,1991.

TECHNICAL UNIVERSITY BUDAPEST, DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE, GROUP FOR COMPUTER SCIENCE AND INFORMATION THEORY

E-mail address: balla@inf.bme.hu

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

ON FACTORIZATION OVER $k((z))[\delta]$

A. BLAGA

Abstract. The article is conceived to have a background in order to obtain an algorithmic method for the formal solutions of a linear differential equation. The solving method is based on a factorization of the differential operators, proposed by using the Newton polygon of a linear differential operator. A subclass of this class of equations is completely solved in the end of the paper.

1. Introduction

Suppose we want to solve a linear differential equation with coefficients in $k(z)$

$$a_n y^{(n)} + a_{n-1} y^{(n-1)} + \cdots + a_1 y' + a_0 y = 0. \tag{1}$$

We assign to (1) the differential operator:

$$D = a_n \frac{d^n}{dz^n} + a_{n-1} \frac{d^{n-1}}{dz^{n-1}} + \cdots + a_1 \frac{d}{dz} + a_0. \tag{2}$$

After factorizing (2) will be much easier to solve (1). Because the polynomial in $\frac{d}{dz}$ described by (2) is not in a commutative ring, the usual Hensel lifting cannot be performed.

Example. Let be the second order linear differential equation

$$y'' - \frac{1}{z+1/2} y' = 0, \tag{3}$$

with 1, $z + z^2$ as solutions (they are not unique).

One can assign the operator

$$D = \frac{d^2}{dz^2} - \frac{1}{z+1/2} \frac{d}{dz}.$$

Try to factorize it!

Because 1 is a solution of (3) we may think to a factorization of the type $D = D_1 \frac{d}{dz}$. Here $D_1 = \frac{d}{dz} - \frac{1}{z+1/2}$. Thus we have one factorization

$$D = (\frac{d}{dz} - \frac{1}{z+1/2})(\frac{d}{dz}).$$

Received by the editors: October 15, 1996.

1991 *Mathematics Subject Classification.* 68Q40, 12Y05.

1991 *CR Categories and Descriptors.* I.1.2 [Algebraic Manipulation]: Algorithms – algebraic algorithms.

Verify that indeed $z + z^2$ is also a solution of the factored equation.

An important remark to make here is that $D_1 = \frac{d}{dz} - \frac{1}{z+1/2}$ and $D_2 = \frac{d}{dz}$ will not commute. This means that if we rewrite $D = D_2 D_1$, the solutions 1 and $z + z^2$ will not verify $D \equiv 0$ anymore. Moreover $D = D_1 D_2$ is not the unique factorization. For example

$$D = (\frac{d}{dz} + \frac{2z+1}{z^2+z} - \frac{2}{2z+1})(\frac{d}{dz} - \frac{2z+1}{z^2+z})$$

gives another decomposition of D. Check the solutions!

2. The Newton polygon of a linear differential operator

We will try to give a factorization of a special class of differential operators, using the Newton polygon and its properties.

Notations. Let be k an algebraically closed field, with $char(k) = 0$ and define $\mathbf{D} = k((z))[\delta]$ a linear differential operator with Laurent series in z as coefficients. Here $\delta = z\frac{\partial}{\partial z}$ is defined to preserve the powers of z:

$$L = \sum_{i=0}^{n} a_i \delta^i, \quad a_i \in k[[z]], \; a_n \neq 0, \quad a_i = \sum_{j=-\infty}^{\infty} a_{ij} z^j.$$

If $a_n = 1$ we say that the differential operator in δ is *monic.*

Remark 2.1. $\delta z = z\delta + z$ *(see the noncommutativity).*

Definition 2.2. *For* $m \in \mathbf{Z}$, $n \in \mathbf{N}$ *call* $z^m \delta^n$ monomial.

Definition 2.3. *The* order *on the monomials:*

$$z^{m_1}\delta^{n_1} \geq z^{m_2}\delta^{n_2} \Leftrightarrow (m_1 \geq m_2, n_1 \leq n_2).$$

Note that this is a partial ordering.

Remark 2.4. *Define* $(n_1, m_1) \geq (n_2, m_2) \Leftrightarrow (n_1 \leq n_2, m_1 \geq m_2)$. *Thus* $z^{m_1}\delta^{n_1} \geq z^{m_2}\delta^{n_2} \Leftrightarrow (n_1, m_1) \geq (n_2, m_2)$.

Definition 2.5. *The* Newton polygon $N(L)$ *of a linear differential operator* $L \in \mathbf{D}$ *is the convex hull of the set*

$$W = \{(x, y) \in \mathbf{R}^2 | \exists z^m \delta^n \in L, (x, y) \geq (n, m)\}.$$

Denote by $(m_1, n_1), (m_2, n_2), \ldots, (m_r, n_r)$ *the vertices of the Newton polygon, including here the special point* $(m, 0)$. *A slope of a Newton polygon is given by* $k_i = \frac{m_{i+1} - m_i}{n_{i+1} - n_i}$ *and the length of the slope* k_i *is* $n_{i+1} - n_i$.

Now, we can define a partial ordering for differential operators of $\mathbf{D}$ and we say that $L_1 \geq L_2$, $L_1, L_2 \in \mathbf{D}$ if all the terms of L_1 are inside of the Newton polygon of L_2.

Example 2.6. *Mark van Hoeij's example.*

$$\begin{aligned} L \;=\; & 7z^{-5} && +2z^{-6}\delta && +2z^{-5}\delta && +3z^{-5}\delta^2 \\ & -3z^{-5}\delta^3 && +5z^{-4}\delta^3 && +z^{-4}\delta^5 && +2z^{-2}\delta^5 \\ & +2z^{-3}\delta^6 && +3z^{-2}\delta^7 && +2z^{-1}\delta^8 && +\delta^9. \end{aligned}$$

We see that there are no negative slopes allowed, but if our ring were commutative we got the negative slope.

Definition 2.7. *The vertices in the Newton polygon are called* extremal points.

The idea of factoring a linear differential operator is contained in the slopes of the Newton polygon; permuting slopes gives other factorizations and moreover, the factors are different for each permutation of slopes.

Definition 2.8. *Let $b(L)$ be the graph of $N(L)$. The* boundary part *of L is:*

$$B(L) = \sum_{(n,m)\in b(L)} a_{n,m} z^m \delta^n.$$

Example 2.9. *In the example above:*

$$B(L) = 2z^{-6}\delta + z^{-4}\delta^5 + 2z^{-3}\delta^6 + 3z^{-2}\delta^7 + 2z^{-1}\delta^8 + \delta^9.$$

Notation. $R(L) := L - B(L)$ and it is called the *interior part* of L.

Remark 2.10. *$R(L) > B(L)$ and $R(L) > L$.*

Proposition 2.11. *If $M_1 = z^{m_1}\delta^{n_1}$, $M_2 = z^{m_2}\delta^{n_2}$ then*

$$M_1 M_2 = z^{m_1+m_2}(\delta + m_2)^{n_1}\delta^{n_2} \tag{4}$$

$$B(M_1 M_2) = z^{m_1+m_2}\delta^{n_1+n_2}. \tag{5}$$

Proof. First part can be done by induction and it is left as an exercise. For the second part we expand the term $(\delta + m_2)^{n_1}$ and since the powers of δ are decreasing from $\delta^{n_1+n_2}$ it is obvious that the boundary part of $M_1 M_2$ will be $z^{m_1+m_2}\delta^{n_1+n_2}$. □

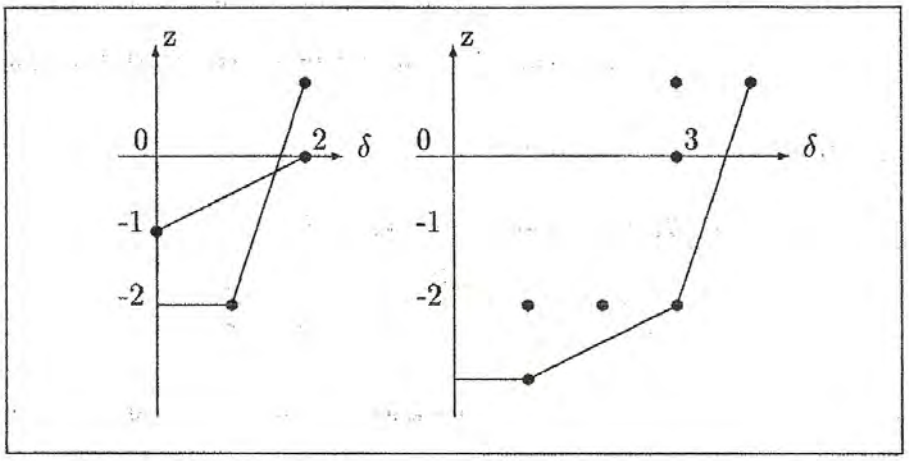

FIGURE 1. Example

Example 2.12. *Let be two operators* $L_1 = z^{-1} + \delta^2$ *and* $L_2 = z^{-2}\delta + z\delta^2$. *Then*

$$L = L_1 L_2 = z^{-3}\delta + \delta^2 + z^{-2}\delta^3 - 4z^{-2}\delta^2 + 4z^{-2}\delta + \\ + 2z\delta^3 + z\delta^4.$$

The interesting thing here are the slopes of L: they include all the slopes of L_1 and L_2, the length of slopes in $L_1 L_2$ is the sum of lengths of the same slopes in L_1 and L_2 and moreover, the Newton polygon of L is the sum of Newton polygons of L_1 and L_2. We will prove this in the next lemma.

Lemma 2.13. *Let be* $L_1, L_2 \in \mathbf{D}$. *Then*

$$N(L_1 L_2) = N(L_1) + N(L_2). \tag{6}$$

Proof. Take

$$L_1 = \sum_{i,j} a_{i,j} z^j \delta^i, \quad L_2 = \sum_{i,j} b_{i,j} z^j \delta^i.$$

Then

$$L_1 L_2 = \sum_{\substack{(i_1, j_1) \\ (i_2, j_2)}} a_{i_1,j_1} b_{i_2,j_2} z^{j_1+j_2} (\delta + j_2)^{i_1} \delta^{i_2}.$$

The terms $z^{j_1+j_2}(\delta + j_2)^{i_1}\delta^{i_2}$ are contained in the Newton polygon $N(L_1) + N(L_2)$. Thus $N(L_1 L_2) \subseteq N(L_1) + N(L_2)$.

Now show that extremal points of $N(L_1)+N(L_2)$ are in $N(L_1L_2)$. So, if (s_2,s_2) is extremal in $N(L_1)+N(L_2)$ then there exist unique $(i_1,j_1)\in b(L_1)$ and $(i_2,j_2)\in b(L_2)$ such that

$$(s_1,s_2)=(i_1,j_1)+(i_2,j_2).$$

Hence $z^{j_1+j_2}\delta^{i_1+i_2}\in N(L_1L_2)$. From this it follows the other inclusion, $N(L_1)+N(L_2)\subseteq N(L_1L_2)$. □

Remark 2.14. *The boundary part of* $L=\sum_{\substack{(i_1,j_1)\in b(L_1)\\(i_2,j_2)\in b(L_2)}} a_{i_1,j_1}b_{i_2,j_2}z^{j_1+j_2}\delta^{i_1+i_2}$ *is*

$$B(L)=\sum_{(s_1,s_2)\in b(L_1L_2)}\left(\sum_{(i_1,j_1)+(i_2,j_2)=(s_1,s_2)} a_{i_1,j_1}b_{i_2,j_2}\right)z^{s_2}\delta^{s_1}. \tag{7}$$

Take this example, now:

$$\begin{aligned} L_1 &= z^{-1}+\delta+terms\ "inside"\ N(L_1),\\ L_2 &= z^{-1}\delta+\delta^2+terms\ "inside"\ N(L_2). \end{aligned}$$

The extremal points for L_1 are $(0,-1),(1,0)$; for L_2 are $(1,-1),(2,0)$.

$$L_1L_2=(z^{-2}-z^{-1})\delta+2z^{-1}\delta^2+\delta^3+terms\ "inside"\ N(L_1L_2).$$

Here, the extremal points are $(1,-2),(3,0)$. But the all combinations of monomials used in (7) are not exhausted. For example we also have

$$\begin{aligned} (0,-1)+(2,0) &= (2,-1)\\ (1,0)+(1,-1) &= (2,-1)\ (see\ point\ P). \end{aligned}$$

So, if (s_1,s_2) is given in a not unique way, then exists common slopes of L_1 and L_2. Moreover, (s_1,s_2) is not an extremal point.

Corollary 2.15. *If* $L_1,L_2\in\mathbf{D}$ *then:*

(*i*) *The set of slopes of* L_1L_2 *is the union of the slopes of* L_1 *and* L_2.

(*ii*) *The length of slopes in* L_1L_2 *is the sum of lengths of the same slopes in* L_1 *and* L_2.

Proof. (*i*) By (6) and geometry of $N(L_1)+N(L_2)$.

(*ii*) Using again (6) and the result of (*i*). □

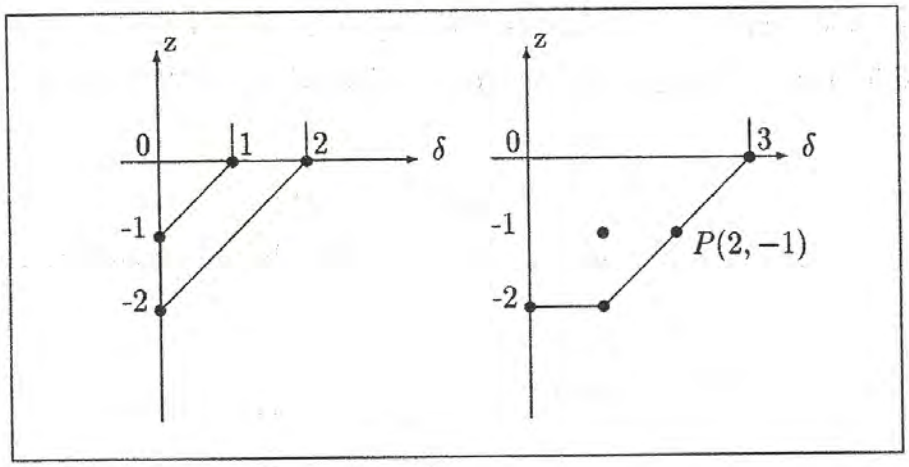

FIGURE 2. The example from the **Remark**

Example 2.16. *Recall the example from the begining of this paper:* $L = \frac{d^2}{dz^2} - \frac{1}{z+1/2}\frac{d}{dz}$, *the linear differential operator of equation* (3). *We have given there two different factorizations:*

$$L = L_1'L_2',\ L_1' = \frac{d}{dz} - \frac{1}{z+1/2},\ L_2' = \frac{d}{dz}$$

$$L = L_1''L_2'',\ L_1'' = \frac{d}{dz} + \frac{2z+1}{z^2+z} - \frac{2}{2z+1},\ L_2'' = \frac{d}{dz} - \frac{2z+1}{z^2+z}.$$

Rewriting everything in terms of δ we obtain

$$\begin{aligned}
L_1' &= z^{-1}\delta - \frac{1}{z+1/2} = \frac{1}{z+1/2}((1+\frac{1}{2}z^{-1})\delta - 1),\\
L_2' &= z^{-1}\delta,\\
L_1'' &= \frac{1}{z(2z+1)(z+1)}((2z^2+3z+1)\delta + 2z^2+2z+1),\\
L_2'' &= \frac{1}{z+z^2}((1+z)\delta - 2z - 1).
\end{aligned}$$

Let's state an important theorem, which can be proved by building up from the simpler cases.

Theorem 2.17. *Let be* $L \in \mathbf{D}$ *with its Newton polygon,* $N(L)$ *having the slopes* $k_1, k_2, \ldots, k_r$. *Then there is a factorization*

$$L = L_1L_2\cdots L_r, \tag{8}$$

where $N(L_i)$ *has unique slope* k_i, *for each* $i \in \{1,2,\ldots,r\}$.

Case of one slope

We will start first from $L = L_1M$, $M \in \mathbf{D}$, where L_1 has only one slope. The simplest case here is that one when the slope is 0. One can write for an $L \in \mathbf{D}$, $L = \sum_{i=-\infty}^{\infty} z^i L(i)(\delta)$, where $L(i) \in k[\delta]$ are polynomials in δ.

Take $n_1 \neq 0$ to be the length of the slope 0 of $N(L_1)$ and we want $L_1 = \sum_{i\geq 0} z^i L_1(i)(\delta)$, $L_1(0)$ is monic of degree n_1 and $L_1(i)$ has degree less than n_1, to maintain the unique slope 0. Also $L_2 = \sum_{i\geq m_1} z^i L_2(i)(\delta)$, where m_1 is the greatest degree occuring in L_1. Successively one can get

$$\begin{aligned} z^{-j}L_1(i)(\delta)z^j &= z^-j\sum_k a_{i,k}\delta^k z^j = z^{-j}z^j\sum_k a_{i,k}(\delta+j)^k = \\ &= \sum_k a_{i,k}(\delta+j)^k = L_1(i)(\delta+j). \end{aligned}$$

Now we compute the product L_1L_2

$$\begin{aligned} L_1L_2 &= (\sum_{i\geq 0} z^i L_1(i)(\delta))(\sum_{j\geq m_1} z^j L_2(j)(\delta)) = \\ &= \sum_{i\geq 0} z^i \sum_{j\geq m_1} z^j z^{-j} L_1(i)(\delta) z^j L_2(j)(\delta) = \\ &= \sum_{i\geq 0} z^i \sum_{j\geq m_1} z^j L_1(i)(\delta+j) L_2(j)(\delta) = \\ &= \sum_{k\geq m_1} z^k \sum_{\substack{i+j=k \\ i\geq 0 \\ j\geq m_1}} L_1(i)(\delta+j)L_2(j)(\delta). \end{aligned}$$

But $L = \sum_{k\geq m_1} z^k L(k)(\delta)$, so

$$\sum_{k\geq m_1} z^k L(k)(\delta) = \sum_{k\geq m_1} z^k \sum_{\substack{i+j=k \\ i\geq 0 \\ j\geq m_1}} L_1(i)(\delta+j)L_2(j)(\delta)$$

For $k = m_1$ we have $L_1(0)(\delta+m_1)L_2(m_1)(\delta) = L(m_1)(\delta)$. For $k = m_1+1$, $L_1(0)(\delta+m_1+1)L_2(m_2+1)(\delta) + L_1(1)(\delta+m_1)L_2(m_1)(\delta) = L(m_1+1)(\delta)$. The last identity is nothing else but the division of $L(m_1+1)(\delta)$ by $L_1(0)(\delta+m_1+1)$, with the remainder $L_1(1)(\delta+m_1)L_2(m_1)(\delta)$ and the quotient to be $L_2(m_2+1)(\delta)$ and by the uniqueness of

the division theorem $L_1(\delta + m_1)$ and $L_2(m_1 + 1)$ are uniquely determined. Doing by this way one can find the factorization of L, by lifting from $L(0) = L_1(0)L_2(m_1)$.

Let's start to say something about the case when the one slope of $N(L_1)$ is > 0. The ideea of this case is that it reduces to the previous case. Take this one slope to be the minimal slope of L and that is $m = b/a$, $(a, b) = 1$, $a, b \in \mathbf{Z}$. Replace δ by $\Delta = t^b\delta$, $z = t^a$. $\Delta = t^b\delta = \frac{t^{b+1}}{a}\frac{d}{dt}$. Thus $\Delta t = \frac{t^{b+1}}{a} + \frac{t^{b+2}}{a}\frac{d}{dt} = \frac{t^{b+1}}{a} + t\Delta$. The new Newton polygon will have one slope 0.

Example 2.18.

$$L = z\delta^2 - 1. \tag{9}$$

Take $m = 1/2, t = z^{1/2}, \Delta = t\delta$. So $\Delta t = \frac{t^2}{2} + t\Delta$ and $\Delta^2 = (t\delta)(t\delta) = t^2\delta^2 + \frac{1}{2}t\Delta$. Thus $t^2\delta^2 = \Delta^2 - \frac{1}{2}t\Delta$. Replacing $t^2\delta^2$ in (8) will give

$$L = \Delta^2 - \frac{1}{2}t\Delta - 1.$$

And thus the unique slope is zero.

References

[1] M. van Hoeij, *Formal solutions and factorization of differential operators with series coefficients*, Manuscript, Holland, 1995.

[2] A.H.M. Levelt, *Polynomials over finite fields*, Course at Uiversity of Nijmegen, Nijemgen, 1995.

[3] B. Malgrange, *Sur la reuction formelle des equations differentielles a singularites irregulieres*, Manuscript, France, 1979.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: blaga@cs.ubbcluj.ro

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

THE DEVELOPMENT OF THE CONCEPT SYSTEM

P. ANDRÁS

Abstract. The concepts are basic bricks of the cognitive processes. The development of the concept system is a very challenging problem. Theories dealing with this problem can be used in various solutions of the human speech processing and written text understanding. In this paper three individual and one communitarian concept system development process are presented. A formal / theoretic framework will be build, by which all of the presented processes can be modeled. Finally an application example is presented.

1. Introduction

The concepts are basic bricks of the cognitive processes. They play a very important role in the understanding of the speech, the reading of the text or in generating intelligent and rational decisions ([2, 3, 4, 5]).

The development of the concept system is a very challenging problem. Theories dealing with this problem can be used in various solutions of the human speech processing and written text understanding. The development of the concept system means the process within which the individual develops its own network of concepts.

In this paper three individual and one communitarian concept system development process are presented. A formal/theoretic framework will be build, by which all of the presented processes can be modeled. Finally an application example is presented.

2. Concept System Development Processes

In this section some concept system development processes will be presented, which are based on natural processes ([6]).

We have to distinguish two types of such processes, the micro level processes, and the macro level processes. We will call micro level processes, those development processes, which are taking part separately within an individuals concept system. The macro level

Received by the editors: October 27, 1996.

1991 *Mathematics Subject Classification.* 68T30, 68T05, 92J40.

1991 *CR Categories and Descriptors.* I.2.6. [Artificial Intelligence]: Learning – concept learning.

processes are those development processes which are working in groups or communities of individuals. The macro level processes works as standardization processes, which set up common meanings of the individually developed concepts.

We will suppose that the concepts are organized in a network ([1, 4]). Each concept will be represented by its name, a set of attributes and a set of behaviors. The attributes are representing the important features of the concepts realizations. The behaviors are specific processes which can modify the attributes of the concept. Each attribute of a concept is a virtual realization of another concept. The concept network is organized along the parent - children relations, which link the concepts ([1]). A concept is the child of some other concepts, if it was developed based on those concepts, inheriting some of their attributes and behaviors. We will use in addition the notion of the pre-concept, which is a structure, which has attributes and behaviors, but not yet has a name and not is inserted as a concept in the concept network. The pre-concept can have multiple attributes, which are attributes with different names, but which are designing the same feature, or it can have behaviors, which are can act contrary in the same time.

The first micro process is describing the process, when the increasing importance of an object or of a phenomena results the creation of a new concept. The steps of this process are the following:

Step 1. *The individual observes the some features of the phenomena (ex. there are a lot of symbols, with a specific meaning, these symbol strings are transformed into machine code, etc.)*

Step 2. *The importance of the observed phenomena is increasing (the cod strings became important, there is a lot of discussion about them), and there is forming a pre-concept, which contains most of the attributes of the phenomena and the collection of main behaviors of it.*

Step 3. *The pre-concept is inserted in the concept network as a descendent of some existing concepts. First there is a searching for existing concepts, which has similar attributes and behaviors like the pre-concept. The pre-concept is inserted as the descendent of the found concepts, inheriting some new attributes and behaviors, too. The new concept gets a new name or gets the name or modified name of one of its ancestors. (It is starting to use the name of source code for the designation of the symbol strings.)*

Step 4. *Some new concepts are inserted in the concept network, which are descendants of the previously inserted concept. The new concepts are realized by the elimination of non- applicable attributes and / or behaviors.*

The second micro level process is describing the case, when a new concept is formed as the consequence of the use of a phrase (some words which are used together). The steps of this process are the following:

Step 1. *A word combination is used, which cannot be interpreted correctly with the existing meaning of the concepts belonging to these words. (ex. it is said that the pipe is burning, but we know that the pipe is not burning normally, only eventually, if it is put in the fire, but this is not the normal use of the pipe)*

Step 2. *To interpret the word combination it is searching for a linkage between the concepts. (It is found that the concept of pipe has an attribute with the name pipe- tobacco, which is a virtual realization of the concept of tobacco, which has the behavior is-burning, by which can be found the interpretation of the word combination.) The linkage can be done through a multiple linkage, too.*

Step 3. *The word combination gains importance, and it is used frequently. As a result is formed a new concept as a descendent of the active concept, which has a new behavior. The new concept has the same name as his ancestor, and the new behavior is realized as the direct interpretation of the behavior designated by the word combination. (The new concept will have the name pipe, and it will have the new behavior of is-burning, which represents the process when the tobacco in the pipe is burning.)*

The third micro process is more simple, and it represents the process of the conciouss learning. Its steps are:

Step 1. *A new concept is presented to the individual, indicating its attributes, behaviors and links within the concept system.*

Step 2. *The new concept is inserted in the concept network at the indicated position.*

Finally it is described a single macro level process, the standardization of the concepts, by which the concepts of the individuals are transformed in a way which results increasing similarity between the individual concepts. The common interpretations are organized in an abstract concept network, which is used with normative force. Based on

this abstract network is realized the conscious learning of the concepts. The steps of the macro process are:

Step 1. *A new concept is developed individually by some individuals who are communicating frequently among them. (Some programmers are developing the concept of functional-programming.)*

Step 2. *Within the community, the members are agreeing about the common sense of the concept, and they start to use it with its common meaning. (The programmers are agreeing about the common meaning of the concept of functional-programming.)*

Step 3. *Other individuals, who didnt knew the concept are learning it with its common meaning. (The students are learning the concept of functional-programming with its common meaning.)*

3. A formal model

In this section we will use the notion of the object for the realization of a concept. Let us consider some preliminary notations:

let $\mathbf{C}$ be the set of concepts,

let $C \in \mathbf{C}$ be a concept,

let note with $r(C)$ the set of objects (realizations) of the concept C,

if R is an object of concept C, then we have $R \in r(C)$,

let us note by $C_1 > C_2$ the descendency relation of C_2 with C_1,

let $C = (N, A, a, B, b)$ be a concept, where

N is the name of the concept (a string of symbols),

$A = (A_1, A_2, ..., A_n)$ is the set of attributes,

$a = (a_1, a_2, ..., a_n)$ is the set of the importances of the attributes,

$B = (B_1, B_2, ..., B_m)$ is the set of behaviors,

$b = (b_1, b_2, ..., b_m)$ is the set of the importances of the behaviors.

The importance of an attribute or of a behavior shows, how prototypic is that attribute or behavior, in other words, how important, or how frequent is the presence of that feature in the objects of the concept. The importances can be interpreted as exact numbers between 0 and 1, or as fuzzy numbers between 0 and 1.

For each A_i we have $A_i \in r(C_i)$, where $C_i \in \mathbf{C}$. The behaviors B_j are conceived in the form

$$B_j = process(A_{i_1}, A_{i_2}, ..., A_{i_{k_j}}),$$

denoting by this a process, within takes part the mentioned attributes, which realizes some behaviors of them.

Let us note by $\mathbf{L} = (S_1, S_2, ..., S_w, ...)$ the world of events which are realized for an individual, the events will be called scenes.

Let $S_t = (O, o, P, p)$ be a scene, where

$O = (O_1, O_2, ..., O_u)$ is the set of objects which take part in the scene,

$o = (o_1, o_2, ..., o_u)$ is the set of subjective importances of the objects,

$P = (P_1, P_2, ..., P_v)$ is the set of actions (processes) which take part in the scene,

$p = (p_1, p_2, ..., p_v)$ is the set of the importances of the processes,

$O_i \in r(C_i)$ and

$P_j = process(O_{i_1}, O_{i_2}, ..., O_{i_{k_j}})$.

Let us note $S_{t_1} \subseteq S_{t_2}$ if $O_{t_1} \subseteq O_{t_2}$ and $P_{t_1} \subseteq P_{t_2}$.

Let $I(S)$ be the importance of scene S, which determines the importances o_i and p_j , too.

After these introductory notes let us turn to the formal interpretation of processes of the dynamics of the concept system. The micro level rules can be formulated in the following way.

Let $S_{i_j}(S_{i_j} \in \mathbf{L})$ be scenes for which exists $S \subseteq S_{i_j} \forall i_j (j = 1, ..., n)$, and let $S = (O, o, P, p)$. In plus let $I(S_{i_j})$ be increasing (or increasing in trend), as j increases, so if we note by imp_j the importance for j, then we have $imp_j > imp_{j_0}$ if $j > j_0$. Then it is creating a pre- concept C (notation create_p(C)), so that $C = (A, a, B, b)$, where

$A = (A_1, ..., A_n)$, and $A_i, O_i \in r(C_i)$,

$B = (B_1, ..., B_m)$, and $B_j = P_j$,

a and b are determined based on the importances o and p.

Let us define the similarity measure between two concepts or pre-concepts as

$$sim(C_1, C_2) = \sum_{i=1}^{n} \frac{(a_{1i}+a_{2i})}{2} \cdot d_i + \sum_{j=1}^{m} \frac{(b_{1j}+b_{2j})}{2} \cdot e_j,$$

where a_i and b_j are the importances of the attributes and behaviors, d_i and e_j are equal to 1 if the feature is identical in the two concepts, and 0 otherwise.

The similarity it is calculated for each permutation of the numbers (1, 2, ..., n) and is taken as the similarity measure, the maximum.

Let us consider that it was created the pre-concept C. There is a searching for the concepts $C_1, C_2, ..., C_m$ for which we have that

$$sim(C, C_k) > s_0.$$

Let C_0 be the concept for which $sim(C, C_k)$ is maximal. We can define different criterias by which we can select the concepts C_k, which will be the ascendants of the new concept. After the selection is created the new concept $C = (N, A, a, B, b)$, as a descendent of the selected concepts, and we will have

$N = N_0$ or it is created a new name for the new concept,

$A = combination(A, A_1, ..., A_{k_1})$,

$B = combination(B, B_1, ..., B_{k_1})$,

$a = combination(a, a_1, ..., a_{k_1})$,

$b = combination(b, b_1, ..., b_{k_1})$,

where combination means a certain combining rule, which generates non-contradictory results. So, we have that $C \in \mathbf{C}$ and $C_k > C$. After this there are created the concepts $C^i, C > C^i$, where the concepts C^i are creates by the elimination of some features, based on the importance of these.

Using this model can be modeled all of the three micro level processes.

To model the described macro process, we have to consider firstly a number of concept networks, $C_1, C_2, ..., C_q$, which contains the concept C with similar, but a little bit different interpretations. The macro process can be modeled by the calculation of some weighted mean of the concepts from the different concept networks, taking in account the importance or the influencing power of each individual, which holds the concept systems.

4. An example

In this section an application of the previously introduced formal framework is presented, for the case of the formation of the concepts of the small positive integer numbers.

Let us consider the scenes $S_k = (O_k, o_k, P, p)$, with

$O_k = (O_{1k}, ..., O_{nk})$, and $O_{ik} \in r(C_k), P = (P_1), P_1$ meaning the simultaneous existence of O_{ik}, for $i = 1, ..., n$. Let us suppose that the scenes S_k happens frequently and $I(S_k)$ is sufficiently high frequently.

Then we have $C_k \subseteq C_0$ and $r(C_k) \subseteq r(C_0)$, so $O_{ik} \in r(C_0)$, where C_0 is the root concept of the concept system.

Because $I(S_k)$ is frequently high, the pre-concept C is formed with the components

$A = (A_1, ..., A_n), A_i \in r(C_0)$,

$B = (B_1)$,

B_1 representing the process of the simultaneous existence of A_i (a and b are defined with the maximal elements).

As a result the concept $CM, C_0 > CM$, is created with the components $A = A$ and $B = B$ ($a = a$ and $b = b$), and with the name $N = many$. Similarly the concept $CU, C_0 > CU$, is created with the name one, and with the components

$A = (A_1), A_1 \in r(C_0)$, $B = (B_1)$,

B_1 representing the process of unique existence of A_1 (a and b are defined with the maximal elements).

After this there are created the concepts C_2, C_3, with the names two, three, as the scenes with two, three, etc. elements became important for that individual. The names of the classes can be individually specific at the start, but later they are modified conform to the macro level processes.

5. Conclusions

The presented model of the concept system development gave a formal framework within which it is possible the natural - like modeling of these processes.

This model can be used as a part of intelligent agent programs or adaptive user interfaces in order to generate the knowledge background for case specific actions and mutual understanding and collaboration.

By further refinement and development of the presented model it is would be possible to create a general scope framework for knowledge base building for expert systems, intelligent agents and adaptive user interfaces.

References

[1] D.I. Carstoiu, *Sisteme Expert*, Editura ALL, Bucuresti, 1994. (in Romanian)

[2] A. Clarck, *A megismeres epitokovei*, Osiris Kiado, Budapest, 1996. (in Hungarian)

[3] H. Gleitman, *Basic Psychology*, 2nd ed., W.W. Norton & Company, New York, 1987.

[4] M. Miclea, *Psihologie Cognitiva*, Editura Gloria, Cluj, 1994. (in Romanian)

[5] S.G. Shanker, The Decline and Fall of the Mechanist Metaphor, in Born R. (ed.): *Artificial Intelligence. The Case Against*, Routledge, New York, 1987.

[6] S. Szilagyi, *Hogyan teremtsunk vilagot*, Erdelyi Tankonyvtanacs, Cluj, 1996. (in Hungarian)

E-mail address: peter@civitas.org.soroscj.ro