STUDIA
UNIVERSITATIS BABEŞ-BOLYAI

MATHEMATICA

3

1994

CLUJ-NAPOCA

Anul XXXIX 1994

STUDIA
UNIVERSITATIS BABEŞ-BOLYAI

MATHEMATICA

3

Redacţia : 3400 CLUJ-NAPOCA str. M. Kogălniceanu nr.1 • Telefon : 116101

SUMAR-CONTENTS-SOMMAIRE

Aniversări-Anniversaries-Anniversaires

Tehnoredactare computerizată: Marcela Topliceanu

STUDIA UNIV. "BABEŞ-BOLYAI", MATHEMATICA, XXXIX, 3, 1994

The Formal Class Model: an Example of an Object-Oriented Design

Pascal ANDRÉ*, Dan CHIOREAN**, Corina CÎRSTEA** and Jean-Claude ROYER**

Rezumat: Lucrarea descrie principalele caracteristici ale modelului cu clase formale. Acest model orientat-obiect cu clase şi moştenire multiplă este strâns legat de tipurile abstracte algebrice, dar cu o tentă mai operaţională. Pentru acest model este prezentată conceptia comenzii `makeindex`. La sfârşit, cu ajutorul unui exemplu sunt prezentate aspecte în validarea şi implementarea semi-automată a proiectării în cadrul acestui model.

Abstract:
This paper describes the main features of the Formal Class Model. This object-oriented model with classes and multiple inheritance is closed to abstract data types, but has a more operational flavour. Using this model we detail the design of the `makeindex` command. Last, using the above example, we illustrate some features about the validation and the implementation of the design.

Résumé:
Dans ce document nous décrivons les principales caractéristiques du modèle des classes formelles. Ce modèle à objets, classes et héritage multiple est proche des types abstraits algébriques mais avec une orientation plus opérationnelle. Nous présentons la conception de la commande `makeindex` dans ce modèle. Finalement nous illustrons à l'aide de l'exemple quelques aspects concernant la validation et l'implantation semi-automatique de la conception.

1. INTRODUCTION

The professional development of large correct software systems in a systematic, structured and modular way is still a challenge for research and practice in software engineering. In recent years, many software techniques improved the technical standards in software engineering, providing better structuring techniques supporting abstraction and reusability.

Object orientation and formal methods are the main fruitful techniques to produce high quality software. Object-Oriented Design needs formal specifications to make proofs and verification automatically. Object-Oriented Programming is a complete and consistent framework for a software development. Incremental development of classes, reusability and extensibility are the main benefits. Abstraction and formal specification techniques were developed to reinforce safety and reusability.

We propose a unifying model for Object-Oriented Design [1], based on algebraic specifications, which unifies the major concepts of Object-Oriented Programming. The outcomes

* *Équipe de Recherche en Technologie à Objets IRIN - Faculté des Sciences et des Techniques Université de Nantes 2, rue de la Houssinière 44072 Nantes Cédex 03, FRANCE*
** *Laboratorul de Cercetare în Informatică Facultatea de Matematică şi Informatică Universitatea "BABEŞ-BOLYAI" str. M. Kogălniceanu, 1 3400 Cluj-Napoca, ROMÂNIA*
* *This work was supported by GDR de Programmation de C.N.R.S., de France. A short version of this paper was presented at ConTI'94* [1]

of such a model are: designing consistent and complete libraries of classes, supporting reverse engineering, application rewriting, comparing or reusing classes coded in the same language or in different languages. A last benefit is the possibility of teaching Object-Oriented Programming in a more abstract way than by using Object-Oriented Languages. The main steps of the design in our formal model are:

- a first design of the class with consistency and inheritance checking;
- the proof of abstract properties;
- testing with rewriting;
- translation to concrete languages.

This paper is organized as follows: Section 2 describes the main goals and aspects of Object-Oriented Design. Section 3 presents the Formal Class Model regarding Object-Oriented Design. Section 4 is a survey of an example designed using this model. Section 5 presents verification and proof techniques supported by the formal design. Section 6 describes an implementation of the formal classes using a concrete language like Eiffel. The conclusions are presented in Section 7.

Our Formal Class Model verifies the requirements of the Object Core Model Group. Furthermore it allows formal specifications of methods and it has more general rules.

2. OBJECT-ORIENTED DESIGN

Object-Oriented Design is characterized by the development of reusable and robust components, named classes. A class definition must be readable, consistent and extensible. There are presently 21 different object-oriented methods described by OMG's Special Interest Group on Analysis and Design (SIGAD). There are extremely different views of many fundamental concepts concerning analysis and design.

We aim at formally design applications and implement them in Object-Oriented Languages. The class construction must be based on an abstract description of its instances. This allows an incremental development of classes and applications, a better reusability and consistency checking. An abstract definition of classes is independent from concrete languages, therefore several implementation languages are possible.

2.1. Formal specification

Formal specifications are needed for a quality software development. The main benefits are: abstraction (reinforces reusability, simplicity and generality), proofs (design, consistency and completion proofs) and documentation (fundamental to reuse and maintain software).

When integrated with object-oriented techniques, formal methods allow precise specification of the semantics of classes. Of course, in order to assist design, various tools must be defined.

2.2. Correctness and Reusability

Object-orientation enhances modularity of specifications enabling separate parts of a development to be worked separately. These parts can be refined independently, since the correctness of high-level parts of a specification can be proved without knowing the internal details of the low-level specifications that implement its operations. Reuse is aided by the ability to specify systems using inheritance, aggregation and genericity

In order to model the transition from specifications to program implementation, classes and formal specifications have to be related to a notion of correctness. This is formalized in an algebraic theory and hence it enables formal reasoning.

3. DESCRIPTION OF THE MODEL

In this section we present a formal way to describe a class. To differentiate between classes from concrete languages and classes from our model, we will name the last ones *Formal Classes*.

Our model unifies major concepts of Object-Oriented Languages. A Formal Class is an abstraction of a concrete class in a language like C++, Eiffel, CLOS or Smalltalk, and also an algebraic specification, as Abstract Data Type (ADT), with an object orientation. Algebraic axioms define an abstract semantics of the behaviour, whose properties can be checked using term rewriting. The Formal Class Model defines a specification language. It is an answer to the requirements of Object-Oriented Design. Conceptually, a Formal Class specifies the object description and behaviour. Syntactically, it contains an aspect and a set of secondary methods. The aspect part is an abstract description of the kernel behaviour of objects, while secondary methods describe the remaining part of the behaviour. Secondary methods allow us to incrementally extend the behaviour of a class, without modifying the characterization of objects.

3.1. Formal Classes

The information concerning a Formal Class is embedded in a box and includes its name, aspect and secondary methods.

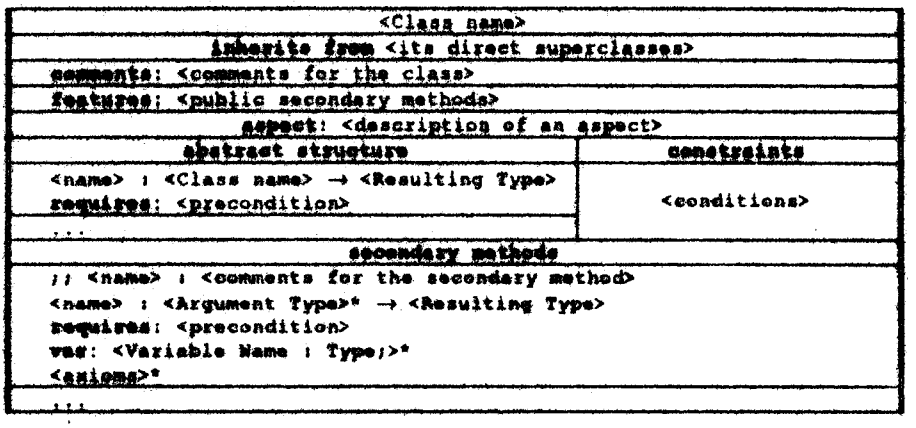


Figure 1: The generic box for a class

We use the following notations: the term `Self` and those declared by `var:` are variables. Other terms beginning with an uppercase letter are classes or predefined types. Terms beginning with a lowercase letter are method names. Message sending is written as a function call: `<selector>(<receiver> <,argument>*)`. The receiver is denoted by `Self` (then there is single dispatch).

A secondary method is described by its profile, axioms, and, if needed, preconditions. We use a functional presentation for methods. Such a presentation is a set of axioms, implicit rewrite-rules (left to right) of form: `condition ⇒ m(Self,...,Z) == u`, where `condition` is a conjunction of equations having the form `t == v` where `t`, `u`, `v` are algebraic terms.

As in Object-Oriented Languages there are abstract classes and abstract methods. The corresponding keyword is ABSTRACT. The class being defined is named the class of interest or the Current Formal Class (CFC). When the resulting type of an operation is the CFC, the operation is called a constructor, else it is called an observer.

3.1.1. Aspect

In this model, object characterization uses the concept of aspect. An aspect is a pair (abstract structure, constraint). The abstract structure is a set of field selectors (partial or total observers of the class). The constraint is a predicate on these field selectors, which can be seen as a condition to create or to modify an object. This is the same idea as a class invariant in Eiffel. Constraints implicitly govern the axioms of the methods. Both the preconditions of field selectors and the constraint are written using conditions, as in algebraic axioms.

3.1.2. Methods

To define a kernel representation of a class, its behaviour is split into two parts: primitive and secondary methods.

Primitive methods are essential for the description and the manipulation of instances. Removing a primitive method causes at least one of the following:

- the set of described instances is modified;
- some parts of an instance can not be accessed;
- instances can not be described or compared.

Primitive methods are twofold: primitive observers and primitive constructors. Among primitive observers we distinguish: the field selectors, the semantic equality (`equal?`) and the description method (`describe`). The set of field selectors is a family of observers which allows to distinguish between two instances of the same class. In this sense we can say that an aspect characterizes a set of objects without confusion. Semantic equality allows us to compare objects in an abstract way (implementation independent). An object description is an external representation of the object.

Primitive constructors are:

- `new`, the generator of instances;
- `copy`, used to create new objects from the existing ones.

In practice, designers construct methods using primitive methods, predefined objects and control structures. Secondary methods are extensions of the primitive ones, that is, every application of a secondary method can be reduced to applications of the primitive methods.

3.2. Relations

Instanciation, inheritance, structural dependency and clientship are the main relations in Object-Oriented Programming. If there are no metaclasses, instanciation is a trivial relation. Clientship (the use relation) is well-known. That's why we do not discuss these two relations.

3.2.1. Structural Dependency

The resulting types of the field selectors ($fsel_i$) are named the structuring types (T_i) of the class. The set of links between a FC and its structuring types defines the Structural Dependency Graph (SDG). A well-designed class defines at least one instance and its instances are finitely generated, so we have a well-found induction on objects. Because of the field selector preconditions ($prec_i$) there is no general and static criterion to check that. In many cases $prec_i$ are equivalent to `true` so a class is well-designed if and only if the SDG is without cycle.

A more general and necessary criterion, but not a completely static one is: a CFC is well-designed if and only if for all $fsel_i:T_i$ we have one of the following:

- T_i is a predefined type,
- or T_i is a well designed FC which does not structurally depend on the CFC,
- or T_i is a FC which structurally depends on the CFC and whose field selector precondition is not equivalent to `true`.

3.2.2. Inheritance and Subtyping

In our model we use inheritance more rigorously than in concrete languages. The instance variables are not inherited. The inheritance rules are:

- Secondary methods are always inherited and it is possible to redefine them.
- There is no inheritance of primitive methods (field selectors, new, etc) or constraints.
- An inheritance link between two classes is possible if every field selector of the superclass exists in the subclass with the same type or a subtype of this type. If there are constraints or field selector preconditions, the rule implies stronger constraints and stronger preconditions in the subclass.

The inheritance graph (IG) must be without cycles. Inheritance implies subtyping. In order to obtain strong typing we add the following rule:

- Methods are redefined according to a rule which is covariant only on the receiver type and the resulting type and other arguments are novariant. This rule is consistent with the previous inheritance criterion and, as we can see in [3], it allows genericity.

To avoid the increase of the complexity in method lookup, name clashes are solved by method redefinition.

3.3. Other Features

3.3.1. Type Checking

The model fits well to dynamically typed languages but also to strongly typed languages like Eiffel. A first problem concerns some terms like `head(tail(newFullPages(...)))` which are meaningful but type erroneous. Our solution to this problem is similar to [6] and described in [3]. This solution needs an additional parsing before the real type checking.

The type checking assumes explicit declarations of variable and method types. We do not handle functions as objects. This avoids the need of a contra-variant rule [5] which would not be consistent with our inheritance rules.

The primitive method profiles for CFC are:

$fsel_i$: CFC → T_i for each field selector
new<CFC> : T_1 ... T_n → CFC
equal? : CFC OBJECT → Boolean
describe : CFC → String
copy : CFC → CFC.

An expression e having the type T is written e:T. A type is either a predefined type (which is not a class) or a FC. The main rule for typing a message expression is:
`let m(e1 .... ek);if ej:Cj, profile(m,C1) = S1 ... Sk → S,`
`and for all j, Cj ake Sj or Cj = Sj, then m(e1 ... ek): S.`
The expression `profile(m,T)` stands for the profile of an operation or a method. If C_1 is a predefined type then m is a predefined operation with a predefined profile.

The type checking algorithm uses the following rules:

- A class is well-typed if its secondary methods are well-typed.
- A method is well-typed if its axioms are well-typed.
- An axiom is well-typed if all its equations are well-typed.
- An equation is well-typed if the left and right expressions are well-typed and have the same type.

If we use a simple covariant redefinition rule, this checking is safe. It means that the evaluation of each well-typed expression built on well-typed classes does not produce a type error. However, it is often useful to use a multi-covariant rule. Problems may arise both in our functional model, and in side effect languages like Eiffel [4]. Note that it is possible to use multiple dispatch; in this case our extended type checking is still safe. With single dispatch and multi-covariant method we have defined additional check to ensure type safeness. Furthermore if such a problem occurs, a very strict additional principle is to systematically redefine methods which directly use a multi-covariant method.

3.3.2. Genericity

As in [9] genericity can be simulated by inheritance. We have defined a formal design for lists and have studied its genericity. We showed in [3] how to create generic lists and how to use them. Usual genericity mechanisms as in Ada, Eiffel or Modula are under study.

3.3.3. Side Effects

Side effects are not an essential concept in OOP, however they are fundamental in practice. The main goals of side effects are some optimizations and the reinforcing of object identity. But the price to be paid is to loose the simple proof techniques of functional programming. The use of side effects allows a soft transition from functional design to real implementation.

Introducing side effects does not modify the inheritance and type checking rules. The model additions are:

- As in imperatives languages, we distinguish statements from expressions. Statements may produce side effects but expressions do not.
- There are additional primitive methods:
  - `modify!` : `CFC` T_1 ... T_n → `CFC` modifies the value associated to a field selector but preserves the identity of the receiver.
  - `eq?` : `CFC OBJECT` → `Boolean` tests the equality of two object identifiers. The differences between `equal?` and `eq?` are classic in Lisp or Scheme.
- Side effects are restricted to the receiver.
- Control structures as `IF THEN ELSE`, `WHILE DO` are possible.

4. AN EXAMPLE OF FORMAL DESIGN

4.1. Description of the Example

Our goal is to design a set of classes, in order to simulate the `\makeindex` command of LaTeX [7]. This command analyses a source text file and produces an index file that contains all the words in the input file, together with the corresponding pages.

Here is an example containing an entry file and the results after applying the `makeindex` command to filein:

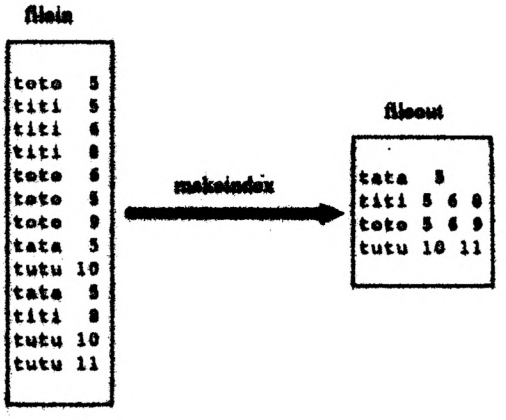


Figure 2: a `\makeindex` example

4.2. The Formal Class Design

From an abstract point of view, both the input and the output file are lists of items, of type `ItemIn` and `Index` respectively. An `ItemIn` is a pair (`String, Integer`): a word and the page in which it appears. There are no constraints concerning the input data. An `Index` is a word followed by a non-empty list of integers: (`String, (Integer, List[ Integer] )`). The informal restriction for the output data is: both the words of the output file and the pages of each index are sorted. A summary of the main FCs is given below:

- `ItemIn`: a pair <word, page>;
- `In`: the input file (a sequence of `ItemIn` instances);
- `Pages`: a sorted list of pages;
- `Index`: a pair <word, its sorted list of pages>;
- `Out`: the output file. An instance of this class is a sequence of `Index` instances.

In the following sections we partially describe some classes. A full description of this example with an algebraic specification, a FC design, the design proof and the Eiffel implementation can be found in Appendix and [10].

In order to design a FC, one must:

- define the abstract structure for the class, and, if needed, provide a constraint;
- verify some criteria concerning the aspect and the inheritance links;
- add secondary methods.

The input data of the \makeindex command consists of items like "word 4".

ItemIn .	
inherits from OBJECT	
comments: class for input data	
aspect : itemin	
features: oneindex	
abstract structure	constraint
word : ItemIn → String	
page : ItemIn → Integer	
secondary methods	

```
;; oneindex : this method transforms an input item in a simple output one
oneindex : ItemIn → Index
oneindex(Self) == newIndex(word = word(Self),
                           pages = add(newEmptyPages(), page(Self)))
```

Figure 3: The ItemIn Formal Class

The ItemIn class is described by its aspect and the set of secondary methods. The field selectors word : ItemIn → String, page: ItemIn → Integer are necessary and sufficient to describe and distinguish the class instances.

4.3. Using Constraints

In order to describe the pages of an Index, we must use a constraint (see Figure 4).

FullPages	
inherits from Pages	
comments: class for non empty list of pages	
aspect : fulllist	
abstract structure	constraint
head : FullPages → Integer	empty?(tail(Self)) orelse
tail : FullPages → Pages	head(Self) < head(tail(Self))

Figure 4 : The aspect of FullPages Formal Class

This constraint states that an instance of FullPages is sorted and without duplication. The instance resulted from a call of newFullPages satisfies this constraint. Applying a secondary method to an instance of FullPages also preserves the constraint (see Section 5.3).

4.4. Describing Secondary Methods

A simple secondary method is oneindex (see Figure 3). A more complex one, where the axioms contain conditions, is insert (see Figure 5).

FullPages
;; insert : insert a page in a full list of pages
insert : FullPages Integer → FullPages
var: X : Integer;
X < head(Self) == true ⇒ insert(Self, X) == add(Self, X)
X = head(Self) == true ⇒ insert(Self, X) == Self
X > head(Self) == true ⇒ insert(Self, X) == add(insert(tail(Self),X), head(Self))

Figure 5 : The insert method of FullPages

If the methods have preconditions, the programmer must ensure that these preconditions are true before using the methods. The idea is the same as in CLU, Modula-3 or Eiffel.

4.5. The Complete Design

It is important to construct the SDG and IG because they allow some simple and useful verifications. The two graphs are given below.

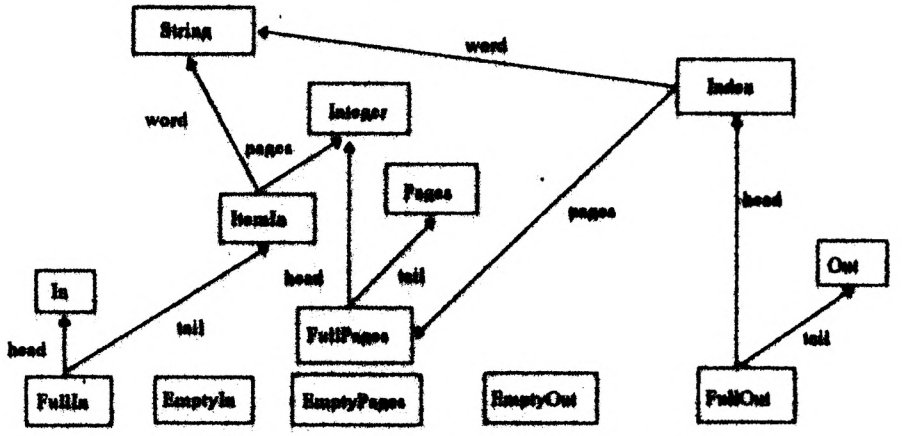


Figure 6: The Structural Dependency Graph

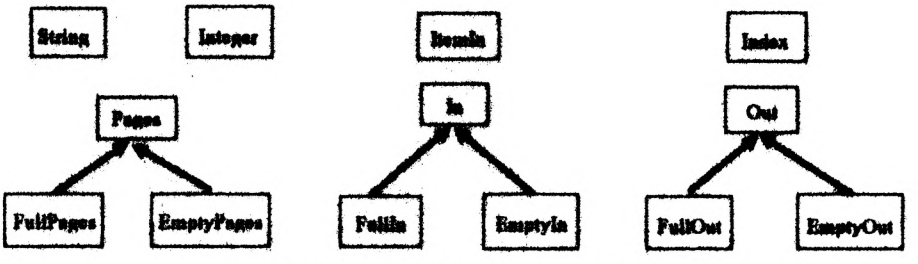


Figure 7: A part of the Inheritance Graph

4.6. Using Side Effects

An example which uses side effects is the putword method given below:

```
                    putword!
                  CLASS Out
// putword! : add an input item
   putword! : Out ItemIn → FullOut
ABSTRACT
                  CLASS EmptyOut
```

```
;; putword! : add an input item
   putword! : EmptyOut ItemIn → FullOut
var: X : ItemIn;
putword!(Self, X) == add(Self, oneindex(X))
```

CLASS FullOut

```
;; putword! : parse a new input item
   putword! : FullOut ItemIn → FullOut
var: X : ItemIn; old, new : Out
BEGIN
  IF word(X) < word(head(Self))
    THEN add(Self, oneindex(X))
    ELSE BEGIN
      data:=Self;
      WHILE not(empty?(data)) andthen word(X) > word(head(data)) DO
        old := data;
        data := tail(data);
      END;
      IF empty?(data) or word(X) < word(head(data))
        THEN modify!(old, tail = add(data, oneindex(X)))
        ELSE modify!(data, head = modify!(head(data),
                     pages = insert!(pages(head(da  )), page(X))))
      END;
  END;
END
```

Figure 8 : The putword! method

4.7. Flat Versus Hierarchical Design

Skill design mixes inheritance and conditional structural dependency. An example of bad design for lists of pages is:

Pages	
inherits from OBJECT	
comments: class for list of pages	
aspect : bad	
abstract structure	constraint
head : Pages → Integer	...
tail : Pages → Pages	

Figure 9 : Bad design for Pages

The following design named "flat" is correct because the general criterion from 3.2.1 is satisfied:

Pages	
inherits from OBJECT	
comments: class for list of pages	
aspect : flat	
abstract structure	constraint
empty? : Pages → Boolean	
head : Pages → Integer requires: empty?(Self) == false	
tail : Pages → Pages requires: empty?(Self) == false	

Figure 10 : Flat design for Pages

However we prefer the "hierarchical" design because we can reuse the FullPages class and this design allows a finer type checking. A complete comparison is out of this paper.

5. VERIFICATION AND PROOFS

5.1. Graph Verifications

As mentioned in 3.2.1., if there are no preconditions for the field selectors, we must check that the SDG is without cycle. We must also verify that the IG is without cycle and that the inheritance criterion is true. Both conditions are satisfied in our example.

5.2. Type Checking Applications

The following short example shows the idea of type checking. Consider the `insert` axiom (see Figure 5):

```
X < head(Self) == true ⇒ insert(Self, X) == add(Self, X).
```

Assume `add: FullPages Integer → FullPages` and `head: FullPages → Integer`. The condition is well-typed because both of its parts have the type `Boolean`. The left and right terms of the equation have the type `FullPages`, so this axiom is well-typed.

To get a right resulting type, the `add` method is redefined in the subclasses of `List`. But this method is multi-covariant, so typing problems may arise. In order to examplify this aspect, let us assume that `SFullList` is a subclass of `TFullList`, corresponding to sorted and without duplication lists of elements of type `S` and `T` respectively, where `S` is a subtype of `T` and instances of both `S` and `T` can be compared using the relation "<". The `TFullList` class can be obtained by replacing the type `Integer` with the type `T` and the type `FullPages` with the type `TFullList` in the `FullPages` formal class. The `SFullList` class is given below. The corresponding `Tlist`, `TEmptyList`, `SList` and `SEmptyList` classes are also defined.

SFullList	
inherits from TFullList	
comments: class for non empty list of pages	
features: add	
aspect: fulllist	
abstract structure	constraint
head : SFullList → Integer	empty?(tail(Self)) orelse head(Self) < head(tail(Self))
tail : SFullList → SList	
secondary methods	
// add : put a new page in the front of the list add : SFullList S → SFullList var: X : S; add(Self, X) == new SFullList(head = X, tail = Self)	

Figure 11. The `SFullList` formal class

Note that the `add` method is redefined by `SFullList`, while the `insert` method is inherited from `TFullList`. Then there is no problem, we can inherit the `insert` method. But a method like `pb(self) = add(Self, newT(*))` in class `TFullList` will be rejected by the type checking. Because using `pb` with a `SFullList` instance produce a type error. Then a solution is to redefine `pb` where `add` is redefined then redefined it in class `SFullList`.

5.3. Proofs

The model allows proofs in an algebraic style. The basic principle is equational deduction or term rewriting. Methods are interpreted as algebraic axioms or rewrite rules. It is trivial for secondary methods and simple for primitive ones [2]. The original thing is the fact that the hierarchy of classes implies a hierarchy of axioms.

We define a call-by-value strategy where method selection depends only on the receiver class. The type of an expression is given by its normal form. An equation is in normal form either if its type is predefined, or if it is a `new<CFC>` on normal form expressions. In the last case its type is simply CFC.

Let `m(e1, ..., en)`. The steps of the evaluation strategy are:

- evaluate all the argument expressions to a normal form.
- the first evaluation, `eval(e1)`, gives the receiver class (if the normal form is a predefined constant and `m` is a predefined operation, then the computation is predefined).
- select the method (`m`) to be applied from the inheritance graph.
- rewrite the entire expression.

Inductive proofs are also possible because we assume a well-found induction on instances, in fact on normal form terms. Consider for example that we want to prove the following lemma:

```
(Self:EmptyPages or Self:FullPages) and X:Integer
⇒ insert(Self, X):FullPages.
```

This means that if `Self` is a list of pages, inserting a new page produces a non empty, sorted list of pages (it verifies the `FullPages` structure and constraint).

Two cases have to be considered:

a) if `Self=newEmptyPages`, the `insert` rule applied is the one in the `EmptyPages` class: `insert(Self,X)==add(Self,X)==newFullPages(head=X,tail=Self)`.

The `FullPages` instance obtained satisfies the constraint:
`empty?(newEmptyPages)==true`.

b) `Self=newFullPages(head=Z,tail=Y)`,
`X:Integer ⇒ insert(Y,X):FullPages`

Now the `insert` method applied is the one in the `FullPages` class (see Figure 5):

- if `X<head(Self)` then `insert(Self,X)=newFullPages(head=X,tail=Self)` and the constraint is true because `head(Self)<head(tail(Self))` by hypothesis.
- if `X=head(Self)` then `insert(Self,X)=Self`, so the receiver does not change.
- if `X>head(Self)` then
`insert(Self,X)=newFullPages(head=Z,tail=insert(Y,X));`

since `tail=insert(Y,X)` is a `FullPages` by induction hypothesis and `Z` is less than `X` and all `Y` pages, then `insert(Self,X)` is a `FullPages` and satisfies the constraint. QED

6. IMPLEMENTATION

Rapid prototyping is an essential tool for specification validation. The transition from FCs to Object-Oriented Programming classes is quite natural and partially automatic. FCs are simple to implement in concrete languages like CLOS, Smalltalk, Eiffel or C++. Such a translation takes the formal description as input and produces the class structure, primitive method code and secondary method signature. In this stage, the concrete secondary methods must be written by hand. However, an automatic translation of secondary methods is possible because of the rewrite rules.

We experimented an automatic translator to Eiffel. The translation begins by associating an Eiffel class (and a file named `<CFC>.e`) to each FC. If the class is abstract, the Eiffel class is DEFERRED. The same holds for abstract methods. All the primitive methods must be specified in the EXPORT clause. For each superclass, an INHERIT clause, with DEFINE and REDEFINE clauses, must be provided. The (re)definitions are used to avoid name clashes.

After that, the main task is to define the FEATUREs. For each field selector which is new or specialized in the subclass, a private attribute and an Eiffel routine which reads this atribute must be defined. The field selector precondition becomes a REQUIRE clause of this routine.

We must also define a CREATE procedure whose argument types are the field selector types. The new<CFC> is a functional call to CREATE. The constraint may become REQUIRE clause for new<CFC>, or better, a class INVARIANT. The primitive equal? is implemented by deep_equal and copy by deep_clone.

Finally, for each secondary method we define an associated Eiffel routine whose profile is the secondary method one, without the receiver type. The translation of axioms must cope with the pointed Eiffel notation: <selector>(<receiver><,args>*) becomes <receiver>.<selector>(<args>*). The precondition is implemented by a REQUIRE clause. Axiom conditions are translated into IF ... THEN ... ELSIF ... END control structures. The result of a method is defined by the special variable RESULT. Note that Self becomes CURRENT and a message like m(Self, *) is translated into m(*). We must use some local variables because CREATE is a procedure, not a function.

Strong typing is not a problem because our type checking is more strict than the Eiffel one.

7. CONCLUSIONS

We defined a minimal abstract model for Object-Oriented Design. This model is a formal specification language, closed to algebraic abstract data types but with an operational flavour. This allows us to adapt the notions of consistency and completion of algebraic specifications. However, the model is often more concrete than algebraic specifications.

We defined rules for inheritance, safe type checking and an abstract semantics based on term rewriting. The inheritance rules allow specialization of the resulting type of a method.

The model is as powerful as the Eiffel language, excepting the association types. Genericity can be simulated by inheritance.

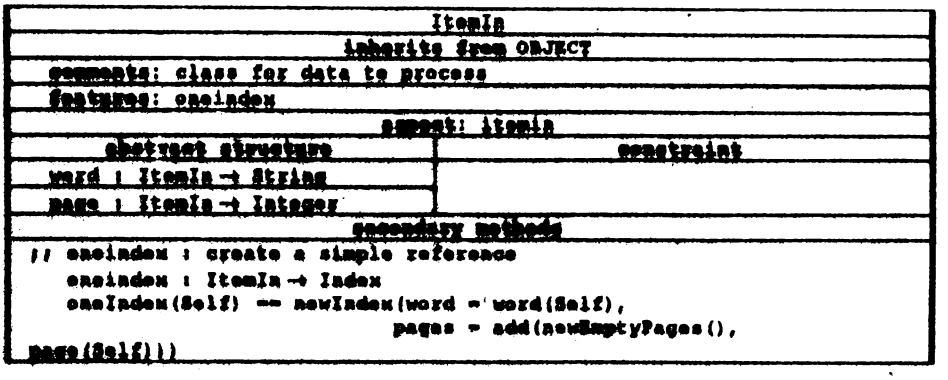
Some extensions are under study: metaclasses (based on the ObjVLisp model), schemes, methods as objects and association types. These extensions add difficulties to type checking.

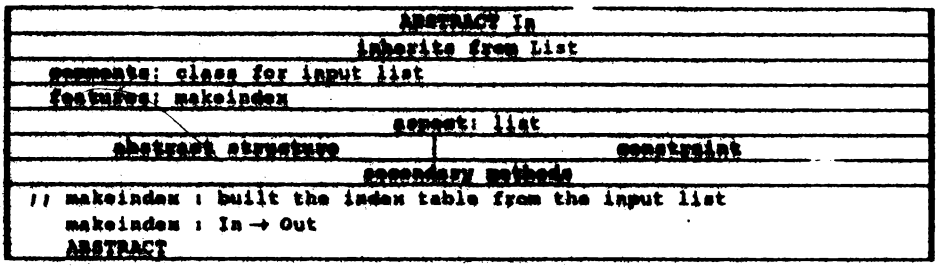
The main features of our model are:

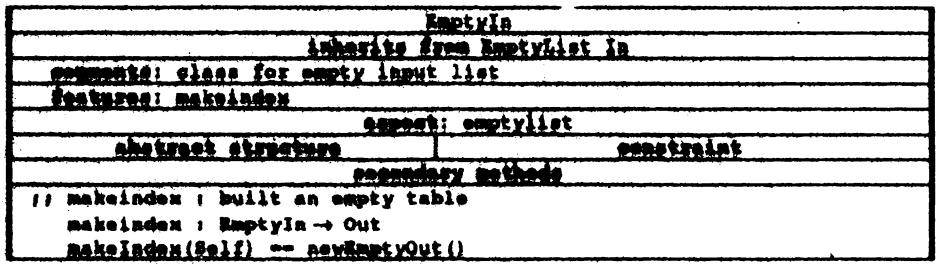
- an object-oriented and formal model to abstractly design applications, i.e. without the need of a particular Object-Oriented Language,
- rules and criterion to check graphs, types, method redefinitions, inheritance links, ...
- a symbolic evaluator and proof technique,
- a direct implementation in concrete languages.

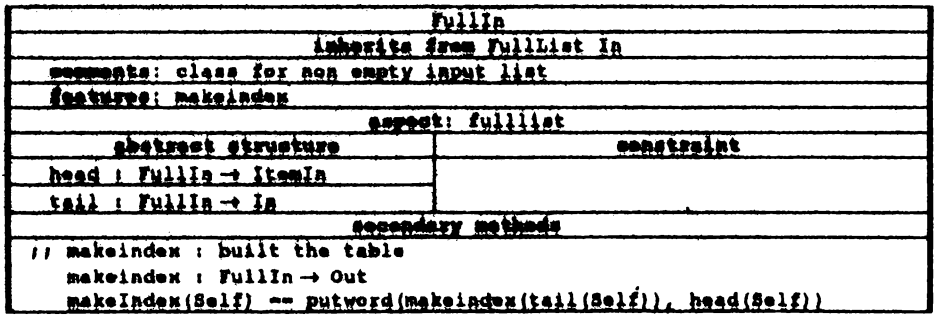
APPENDIX

Formal Classes









ABSTRACT Pages

inherits from List

comments: class for list of pages

features: insert, add

aspect: list

abstract structure	constraint

secondary methods

```
;; add : put a new page in the front of the list
   add : Pages Integer → FullPages
   ABSTRACT
;; insert : insert a new page in the list
   insert : Pages Integer → FullPages
   ABSTRACT
```

EmptyPages

inherits from EmptyList Pages

comments: class for empty list of pages

features: insert, add

aspect: emptylist

abstract structure	constraint

secondary methods

```
;; add : put a new page in the front of the list
   add : EmptyPages Integer → FullPages
   var: X : Integer;
   add(Self, X) == newFullPages(head = X, tail = Self)
;; insert : insert a new page in an empty list
   insert : EmptyPages Integer → FullPages
   var: X : Integer;
   insert(Self, X) == add(Self, X)
```

FullPages

inherits from FullList Pages

comments: class for non empty list of pages

features: insert, add

aspect: fulllist

abstract structure	constraint
head : FullPages → Integer	empty?(tail(Self)) orelse
tail : FullPages → Pages	head(Self) < head(tail(Self))

secondary methods

```
;; add : put a new page in the front of the list
   add : FullPages Integer → FullPages
   var: X : Integer;
   add(Self, X) == newFullPages(head = X, tail = Self)
;; insert : insert a new page in a full list
   insert : FullPages Integer → FullPages
   var: X : Integer;
   X < head(Self) == true ⇒ insert(Self, X) == add(Self, X)
   X = head(Self) == true ⇒ insert(Self, X) == Self
   X > head(Self) == true ⇒ insert(Self, X) ==
                        add(insert(tail(Self), X), head(Self))
```

Index	
inherits from OBJECT	
comments: class for output item	
features: oneindex	
aspect: index	
abstract structure	**constraint**
word : Index → String	
page : Index → FullPages	
secondary methods	
;; makeindex : built the table makeindex : FullIn → Out makeIndex(Self) == putword(makeindex(tail(Self)), head(Self))	

Out	
inherits from List	
comments: class for output table	
features: putword, add	
aspect: list	
abstract structure	**constraint**
secondary methods	
;; add : put a new index in the front of the list add : Out Index → FullOut ABSTRACT	
;; putword : parse a new input item makeindex : Out ItemIn → FullOut ABSTRACT	

EmptyOut	
inherits from EmptyL..t Out	
comments: class for empty output table	
features: putword, add	
aspect: emptylist	
abstract structure	**constraint**
secondary methods	
;; add : put a new index in the front of the list add : EmptyOut Index → FullOut var: X : Index; add(Self, X) == newFullOut(head = X, tail = Self)	
;; putword : add an input item putword : EmptyOut ItemIn → FullOut var: X : ItemIn; putword(Self, X) == add(Self, oneindex(X))	

FullOut	
inherits from FullList Out	
comments: class for non empty output table	
features: putword, add	
aspect: fulllist	
abstract structure	**constraint**
head : FullOut → Index	empty? (tail(Self)) orelse
tail : FullOut → Out	word(head(Self)) < word(head(tail(Self)))
secondary methods	
;; add : put a new index in the front of the list add : FullOut Index → FullOut add(Self, X) == newFullOut(head = X, tail = Self)	
;; putword : parse a new input item putword : FullOut ItemIn → FullOut var: X : ItemIn; word(X) < word(head(Self)) == true ⇒ putword(Self, X) == add(Self, oneindex(X)) word(X) = word(head(Self)) == true ⇒ putword(Self, X) == add(tail(Self), newIndex(word = word(X), pages = insert(pages(head(Self)), page(X)))) word(X) > word(head(Self)) == true ⇒ putword(Self, X) == add(putword(tail(Self), X), head(Self))	

Eiffel Classes

This appendix contains some Eiffel V2.3 classes resulting from a direct translation of formal classes

```
CLASS ItemIn
EXPORT word, page, oneindex
INHERIT OBJECT
FEATURE
    -- private fields
    word_private : String;
    page_private : Integer;

    -- create redefinition
    CREATE (m : String, p : Integer) IS
      DO
      word_private := m;
      page_private := p;
      END; -- create

    -- field selectors
    word : String IS
      DO
      RESULT := word_private;
      END; -- word

    page : Integer IS
      DO
      RESULT := page_private;
      END; -- page

    -- a simple index
    oneindex : Index IS
      LOCAL e : EmptyPages;
      DO
      e.CREATE;
      RESULT.CREATE(word, e.add(page));
      END; -- oneindex
END; -- ItemIn

CLASS Index
EXPORT word, pages
INHERIT OBJECT
FEATURE
    -- private fields
    word_private : String;
    pages_private : FullPages;

    -- create redefinition
    CREATE (m : String, p : FullPages) IS
      DO
      word_private := m;
      pages_private := p;
      END; -- create

    -- field selectors
    word : String IS
      DO
      RESULT := word_private;
      END; -- word

    pages : FullPages IS
      DO
      RESULT := pages_private;
      END; -- pages
END; -- Index

DEFERRED CLASS In
EXPORT makeindex, add
INHERIT List
    REDEFINE add;
FEATURE
    -- built an index table
    makeindex () : Out IS
      DEFERRED
      END; -- makeindex

    -- put an ItemIn in front of the list
    add (i : ItemIn) : FullIn IS
      DEFERRED
      END; -- add
END; -- In

CLASS EmptyIn
EXPORT makeindex, add
INHERIT EmptyList
    REDEFINE add;
    In
    REDEFINE add;
FEATURE
    -- create redefinition
    CREATE IS
      DO
      END; -- create

    -- put an ItemIn in front of the list
    add (i : ItemIn) : FullIn IS
      DO
      RESULT.CREATE(i, current);
      END; -- add

    -- creates an Index
    makeindex () : EmptyOut IS
      DO
      RESULT.CREATE();
      END; -- makeindex
END; -- EmptyIn

CLASS FullIn
EXPORT head, tail, makeindex, add
INHERIT FullList
    REDEFINE head, tail, add;
    In
    REDEFINE add;
FEATURE
    -- private fields
    private_head : ItemIn;
    private_tail : In;

    -- create redefinition
    CREATE (i : ItemIn; l : In) IS
      DO
      private_head := i;
      private_tail := l;
      END; -- create

    -- field selectors
    head : ItemIn IS
      DO
      RESULT := private_head;
      END; -- head

    tail : In IS
      DO
      RESULT := private_tail;
      END; -- tail

    -- put an ItemIn in front of the list
```

```
    add (i : ItemIn) : FullIn IS
      DO
      RESULT.CREATE(i, current);
      END; -- add

    -- creates an Index
    makeindex () : FullOut IS
      DO
      RESULT := private_tail.makeindex
        .putword(private_head);
        END; -- makeindex
END; -- FullIn

DEFERRED CLASS Pages
EXPORT insert, add
INHERIT List
    REDEFINE add
FEATURE

    -- put a new page in front of the list
    add(i : Integer) : FullPages IS
      DEFERRED
      END; -- add

    -- insert a new page in the list
    insert(i : Integer) : FullPages IS
      DEFERRED
      END; -- insert
END; --Pages

CLASS EmptyPages
EXPORT insert, add
INHERIT EmptyList
    REDEFINE add
    Pages
    REDEFINE add

FEATURE

    -- put a new page in front of the list
    add(i : Integer) : FullPages IS
      RESULT.CREATE(i, current);
      END; -- add

    -- insert a new page in the list
    insert(i : Integer) : FullPages IS
      RESULT.CREATE(i, current);
      END; -- insert
END; -- EmptyPages

CLASS FullPages
EXPORT head, tail, insert, add
INHERIT FullList
    REDEFINE head, tail, add
    Pages
    REDEFINE add

FEATURE
    -- private fields
    private_head : Integer;
    private_tail : Pages;

    -- field selectors
    head : Integer IS
      DO
      RESULT := private_head;
      END; -- head

    tail : Pages IS
      DO
      RESULT := private_tail;
      END; -- tail

    -- put a new page in front of the list
    add(i : Integer) : FullPages IS
      RESULT.CREATE(i, current);
      END; -- add

    -- insert a new page in the list
    insert(i : Integer) : FullPages IS
      IF i < private_head
        THEN RESULT.Create(i, current);
        ELSEIF i = private_head
          THEN RESULT := current;
          ELSE RESULT.CREATE(private_head,
            private_tail.insert(i);
      END;
      END; -- insert

INVARIANT
    private_tail.empty? orelse private_head.word <
private_tail.private_head.word;
END; -- FullPages

DEFERRED CLASS Out
EXPORT putword, add
INHERIT List
    REDEFINE add;
FEATURE

    -- put an index in front of the list
    add (i : Index) : FullOut IS
      DEFERRED
      END; -- add

    -- insert a word and its page
    putword (i : ItemIn) : FullOut IS
      DEFERRED
      END; -- putword
END; -- Out

CLASS EmptyOut
EXPORT putword, add
INHERIT EmptyList
    REDEFINE add;
    Out
    REDEFINE putword, add;
FEATURE
    -- create redefinition
    CREATE IS
      DO
      END; -- create

    -- put an index in front of the list
    add (i : Index) : FullOut IS
      DO
      RESULT.CREATE(i, current);
      END; -- add

    -- insert a word and its page
    putword (i : ItemIn) : FullOut IS
      DO
      RESULT := add(i.oneindex, current);
      END; -- putword
```

```
    END; -- EmptyOut

    CLASS FullOut
    EXPORT head, tail, putword, add
    INHERIT FullList
        REDEFINE head, tail, add;
        Out
        REDEFINE putword, add;
    FEATURE
        -- private fields
        private_head : Index;
        private_tail : Out;

        -- create redefinition
        CREATE (i : Index; l : Out) IS
          DO
          private_head := i;
          private_tail := l;
          END; -- create

        -- field selectors
        head : Index IS
          DO
          RESULT := private_head;
          END; -- head

        tail : Out IS
          DO
          RESULT := private_tail;
          END; -- tail

        -- put an index in front of the list
        add (i : Index) : FullOut IS
          DO
          RESULT.CREATE(i, current);
          END; -- add

        -- insert a word and its page
        putword (i : ItemIn) : FullOut IS
          LOCAL ind : Index;
          DO
          IF i.word < head.word
          THEN    RESULT    :=    add(i.oneindex,
current);
          ELSIF i.word.deep_equal(head.word)
            THEN ind.CREATE(i.word,
                 head.pages.insert(i.page);
              RESULT := tail.add(ind);
            ELSE RESULT :=
tail.putword(i).add(head);
          END;
          END; -- putword

    INVARIANT
        private_tail.empty? orelse private_head.word <
        private_tail.private.head.word;
END; -- FullOut
```

REFERENCES

[1] Pascal André, Dan Chiorean Corina CIRSTEA and Jean-Claude Royer, Object-Oriented Design With Formal Classes, in: *ConTI'94: International Conference on Technical Informatics, 1994*, 16-19 November, Timişoara, România.

[2] Pascal André, Dan Chiorean and Jean-Claude Royer, The Formal Class Model, in: *Joint Modular Languages Conference*, Ulm, Germany, (1994).

[3] Michel Augeraud and Jean-Claude Royer, Une interprétation du concept de classe en termes de type abstrait, in: *Journées du GDR Programmation avancée et outils pour l'intelligence artificielle*, pages 13-27, Nancy, France, (1992) Rapport du GRECO de Programmation.

[4] Pascal André and Jean-Claude Royer, La modélisation des listes en programmation par objets, in: Pierre Cointe, Christian Queinnec and Bernard Serpette, eds. *Journnées Francophones des Langages Applicatifs*, Collection Didactique, 11 (1994) 259-285.

[5] William R. Cook. A Proposal for Making Eiffel Type-safe, in: *The Computer Journal*, 32 (4) (1989) 305-311.

[6] Luca Cardelli and Peter Wegner, On Understanding Types, Data Abstraction and Polimorphism, in: *Computing Surveys*, 17(4) (1985) 471-522.

[7] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Megrelis, José Meseguer, and Timothy Winkler, An Introduction to OBJ3, Rapport de recherche 88-R-001, Rapport du Centre de Recherche en Informatique de Nancy, (France, Vandoeuvreles-Nancy, 1988).

[8] Leslie Lamport. LaTeX User's Guide and Reference Manual (Addison-Wesley Publishing Company Inc., 1986).

[9] Kevin Lano and Howard Haughton, eds., *Object-Oriented Specification Case Studies. Object-Oriented Series* (Prentice Hall, 1993).

[10] Bertrand Meyer, Object-Oriented Software Construction, International Series in Computer Science (Prentice Hall, 1988).

[11] Jean-Claude Royer, Un exercice de spécification formelle de preuve et de conception à objets, Rapport de recherche 30, IRIN, Faculté des Sciences et des Techniques, Université de Nantes, 1993.

[12] Pierre Cointe. Metaclasses Are First Classes: The ObjVlisp Model. In ACM OOPSLA'87 Proceedings, 156-167. ACM, October 1987

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

DISTRIBUTED PROCESSING IN EXTENDED B-TREE

Florian Mircea BOIAN* **and Alexandru VANCEA***

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 10, 1995
AMS subject classification: 68Q22, 65Y05, 65Y10

REZUMAT. - Procesarea distribuită în B-arbori extinşi. În această lucrare se arată că structura de B-arbore este o structură de date foarte indicată pentru procesarea sa într-un mediu distribuit în care comunicarea se face prin transmitere de mesaje. În acest context se propun unele tehnici de procesare distribuită în B-arbori, tehnici inspirate de algoritmii clasici de mapare a taskurilor într-un sistem distribuit. Nu poate fi stabilită o tehnică optimă în cazul general, problema în acest caz fiind o problemă NP-hard.

1. Preliminaries

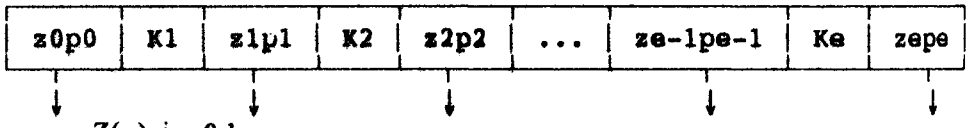
A B-tree was formally defined in [Knuth76]. We denote by **m** the order of the B-tree, and we denote by **e** the number of keys from the current B-tree node. By p, with possible subscripts we denote pointers to B-tree nodes. Finally, by **K**, with possible subscripts, we denote value(s) of key(s) from B-tree. If p is a pointer to a B-tree node, we denote by S(p) the B-subtree having the root in the node pointed by p.

Definition 1. The **possession** of S(p) is defined as the total number of keys from S(p). We denote this number by Z(p).

Let $a = K_{i+1}K_{i+2}...K_{i+r}$ be the word of the r succesive keys from a particular node of B-tree. Let p_i, p_{i+1}, p_{i+2}, ..., p_{i+r} be the neighbour pointers for the keys from a. By S(a) we denote the B-subtree which has in its root only the keys from a and the descendents $S(p_i)$, $S(p_{i+1})$, $S(p_{i+2})$, ..., $S(p_{i+r})$.

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

Definition 2. An **Extended B-tree** [Boian89] is a B-tree having in its nodes the following information:



where $z_i = Z(p_i)$, i = 0,1,...,n.

An example. In figure 1, an extended B-tree is presented. In each node, only the values of keys are presented. For leafless nodes there are two arrows near each key: one on the left and the other on the right. On the left of each row, in brackets, the value of possession appears, and on the right, the value of the pointer (here is the number of the node) appears.

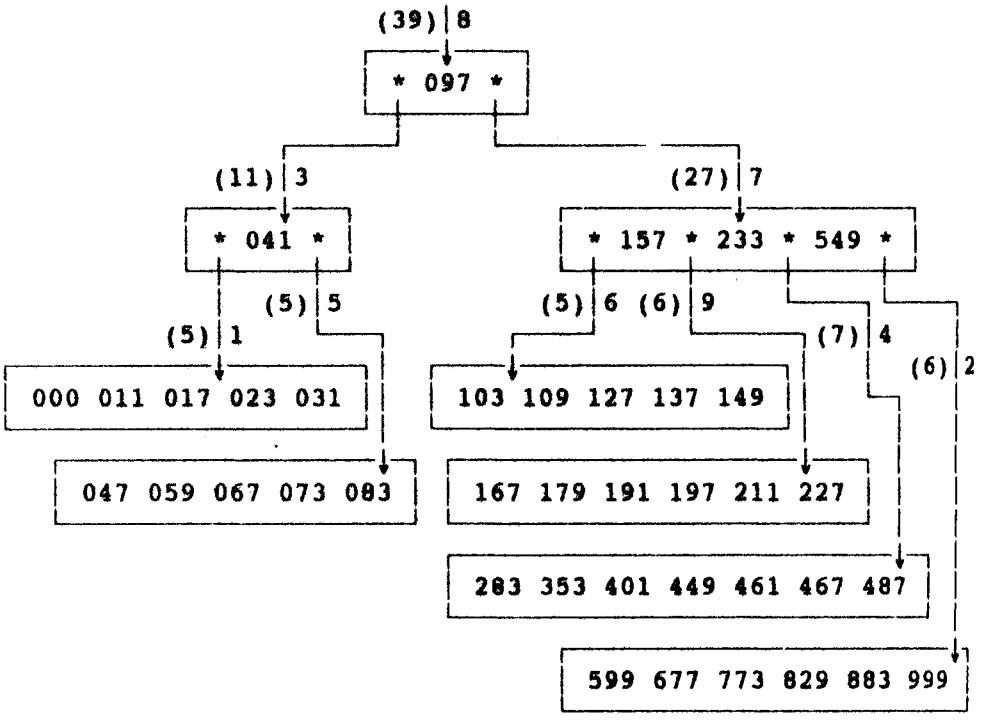


Figure 1. An extended B-tree

For example, S(7) has the nodes 7, 6, 9, 4, 2, and Z(7) = 27. If a = "157 233", then S(a) is S(7) without the key "549" and without the node 2, and Z(a) = 20.

Extended B-tree transformation. The operations with B-tree are presented in [Knuth76]. In [Boian89] and [Boian89a] we have described some ideas to implement an extended B-tree. In figures 2, 3 and 4 three pairs of transformations are presented: rotate left/right, transform a node into two or viceversa and the transformation of two nodes into three or viceversa. In these figures, we denote by lower case (a, b, ..., h, i) the sequences, possibly empty, from consecutive keys (from the same node), and by an uppercase (C, E, G) a key from a node.

When a transformation is applied, the possessions for new nodes must be computed only from the old ones, without considering the other nodes from the B-tree. In the following, for the four usual transformations, the new possessions are:

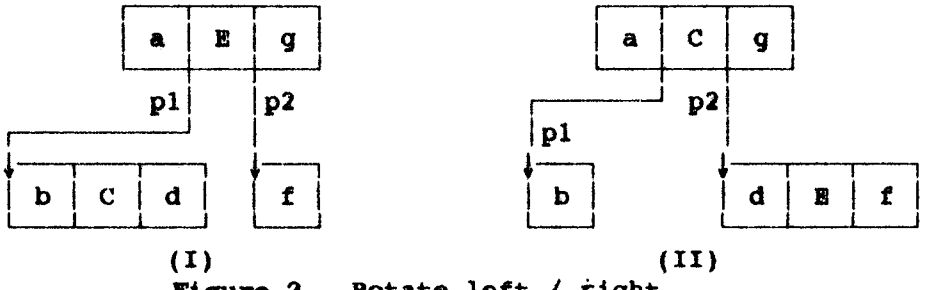


Figure 2. Rotate left / right

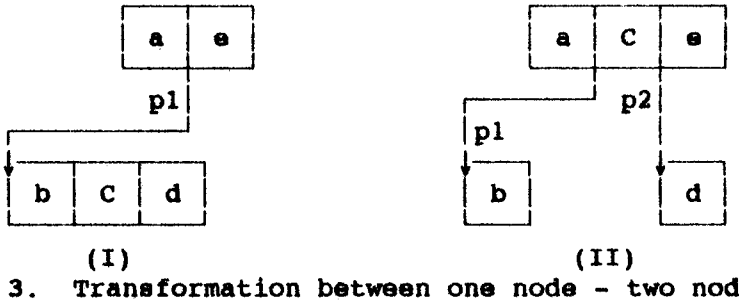


Figure 3. Transformation between one node - two nodes

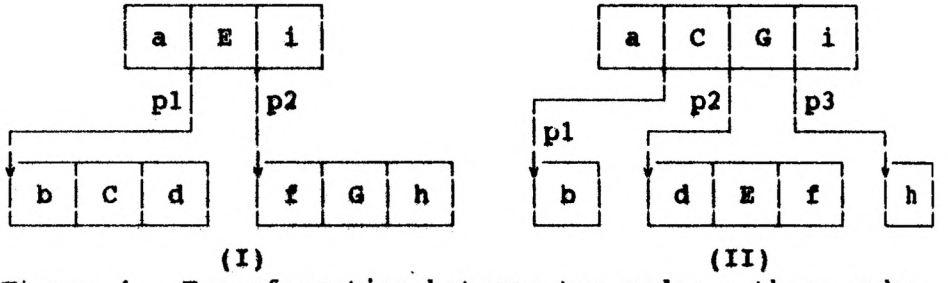


Figure 4. Transformation between two nodes - three nodes

1) From (I) to (II) of the fig.2 (rotate to right):

$Z(p_1) := Z(b)$;

$Z(p_2) := Z(d) + 1 + Z(f)$;

2) From (II) to (I) of the fig.2 (rotate to left):

$Z(p_1) := Z(b) + 1 + Z(d)$;

$Z(p_2) := Z(f)$;

3) From (II) to (I) of the fig.3 (fusion of two nodes into one):

$Z(p_1) := Z(b) + 1 + Z(d)$;

4) From (I) to (II) of the fig.4 (transformation of two nodes into three):

$Z(p_1) := Z(b)$;

$Z(p_2) := Z(d) + 1 + Z(f)$;

$Z(p_3) := Z(h)$.

We have used these four transformations in [Boian89] for implementation. If only these are used, at most two nodes are necessary for operations with the B-tree.

2. Distributed processing techniques for extended B-trees

One of the most important aspects in obtaining good execution efficiency in parallel and distributed computing is *load balancing*. Before executing a program on a distributed

computing system the work to be done must be partitioned among the available processors and for this to be an efficient action this work needs to be *balanced*.

Therefore, having at our disposal a distributed computing system, we consider that an adequate B-tree nodes distribution onto the available network configuration is of an extremely importance.

There are many ways for assigning the B-tree nodes to the available processing elements. We present below some of them, called *source initiated schemes*, which we consider well suited solutions for our problem of obtaining good distributed execution efficiency. In the following, we will identify the necessary activity at each B-tree node with a process, having then the correspondence "a node" - "one process" for simplifying our discussion.

The nature of the transformations that are taking place in a B-tree suggests a *message passing* distributed modelling. For example, the necessary messages for the cases described in figure 2, 3 and 4 are given below. Except create and remove, all the other remaining actions can be executed in parallel:

i). rotate left to right (fig.2, I to II):

p_0 to p_2 sends "E";

p_1 to p_2 sends "d";

p_1 to p_0 sends "C";

ii). rotate right to left (fig.2, II to I):

p_2 to p_0 sends "E";

p_2 to p_1 sends "d";

p_0 to p_1 sends "C";

iii). divide a node from two (fig.3, I to II):

Create a new node p_2;

p_1 to p_0 sends "C";

p_1 to p_2 sends "d";

iv). join the nodes p_1 and p_2 into p_1 (fig.3, II to I):

p_0 to p_1 sends "C";

p_2 to p_1 sends "d";

Remove the node p_2;

v). divide the nodes p_1 and p_2 from three (fig.4, I to II):

Create a new node p_3;

p_0 to p_2 sends "E";

p_1 to p_0 sends "C";

p_1 to p_2 sends "d";

p_2 to p_0 sends "G";

p_2 to p_3 sends "h";

vi). join the nodes p_1, p_2 and p_3 into p_1 and p_2 (fig.4 II to I):

p_3 to p_2 sends "h";

p_0 to p_2 sends "G";

p_2 to p_1 sends "d";

p_0 to p_1 sends "C";

p_2 to p_0 sends "E";

Source initiated schemes are characterized by the fact that the work splitting is performed only when an idle processor (called the *source* in this context) requests some work to do. Hence, the schemes presented here are all demand driven schemes. In all such schemes when a processor runs out of work it generates a request for work. What differentiates all the following different load balancing schemes is the way in which is made the selection of the target for this work request. This selection should be such as to minimize the total number of work requests and to balance the load among processors with fewest possible work transfers. The basic load balancing algorithm is the following [Gra91]:

```
while (not terminated)
        while (work not available)
                determine target;
                send request for work to target;
                receive message if any;
                if (message is work request) send a reject;
                if (message is a reject) reset flag to indicate
                   that a fresh target has to be determined and
                   another request for work be generated;
                service work requests and termination messages;
        end-while
        do work until exhausted and at the same time service
        work requests;
end-while;
```

In the *Asynchronous Round Robin* (ARR) scheme each processor maintains an independent variable *count*. Whenever a processor runs out of work it reads the value of *count* and sends a work request to that particular processor. The value of *count* is incremented (modulo P) each time its value is read and a work request sent. Initially, the value of *count* is set to ((p+1) modulo P) where p is the processor identification number. Since each processor has a counter of its own, work requests can be generated by each processor independent of the other processors.

Global Round Robin (GRR) considers the variable *count* stored in the processor 0 of a hypercube. When a processor needs work it requests and gets the value of this variable and the processor 0 increments the value by 1 modulo P before responding to another request. The processor needing work is sending now a request to the processor whose number was supplied by processor 0. This algorithm ensures that the work requests are uniformly distributed over all processors. A potential drawback of this scheme is the possible competition for reading *count*.

In the *Nearest Neighbour* scheme, a processor running out of work sends a work request to its nearest neighbours in a round robin fashion (for example, on a hypercube a processor will request its logN neighbours). Thus we have locality of communication for both work requests and actual work transfers. For networks in which the distance between any two processors is the same this scheme is the same as the Global Round Robin. In a way, this

scheme can be considered an adaptation of ARR for networks that are not completely connected. A potential drawback of this scheme is that localized concentration of work takes a longer time to globally balance the load among all processors.

To avoid competition for reading *count* in GRR, in *GRR with message combining* all the requests to read the value of *count* at processor 0 are combined at some intermediate processors. Thus, the total number of requests that have to be solved by processor 0 and its neighbours is greatly reduced. This technique is basically a software implementation of the *fetch&add* operation [Ski91].

In the *scheduler based load balancing* scheme a processor is designated as a scheduler. This processor maintains a list of all possible processors which can donate work. Initially this list contains just one processor which has all the work. Whenever a processor goes idle it sends a request to the scheduler. The scheduler then enquires the processors in the list of active processors in a round robin fashion till it gets work from one of the processors. This processor is then placed at the tail of the list and the work received by the scheduler is forwarded to the requesting processor.

Let's notice that the performance of this last scheme can be degraded significantly by the fact that all messages (including messages containing the actual work) are routed to the scheduler. This poses an additional bottleneck for the work transfer. We can improve this scheme so that the poll be still generated by the scheduler but the work be transferred directly to the requesting processor instead of being routed through the scheduler.

3. Conclusions and further reasearch

We presented in this paper some proposals on efficiently distribute the nodes of a B-tree on the processing elements of a distributed computing system. It's difficult to say in the general case which is the best strategy to use, taking into account the fact that the distribution of processes among processors in the general case in known to be NP-hard [Tao92].

As further research we hope to have access at some parallel architectures and make experimental evaluations of these load balancing techniques for a large number of B-tree applications and compare the performances with those theoretically estimated. Also, we want to determine more exactly the most efficient contents of the notion of process in this context,

this paper making the simplifying assumption "one node" - "one process", not necessarily the best choice.

REFERENCES

[Akl90] S.G.Akl - *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[Boian89] Boian F. M. - Sistem de fişiere bazat pe B-arbori, în *Lucrarile celui de-al VII-lea colocviu naţional de informatică*, INFO-IASI, 1989, pp. 33-40.

[Boian89a] Boian F. M. - Căutare rapidă în B-arbori, în *Lucrările simpozionului "Informatica şi aplicaţiile sale"*, Zilele academice Clujene, Cluj-Napoca, 1989.

[Gra91] A.Y.Grama, V.Kumar, V.N.Rao - Experimental Evaluation of Load Balancing Techniques for the Hypercube, in *Parallel Computing '91* (D.J.Evans et al.editors), Elsevier Science Publishers, pp.497-512.

[Knuth76] Knuth D.E. - *Tratat de programarea calculatoarelor*, vol III, Sortare şi căutare. Ed. Tehnică, Bucureşti, 1976.

[Kri89] Krishnamurthy E.V. - *Parallel Processing. Principles and Practice*, Addison-Wesley, 1989.

[Ski91] D.B.Skillicorn - Models for Practical Parallel Computation, *International Journal of Parallel Programming*, vol.20, no.2, pp.133-158, 1991.

[Tan92] A.S.Tanenbaum - *Modern Operating Systems*, Prentice Hall, 1992.

[Tao92] Tao L., Zhao Y.C., Narahari B. - *Efficient Heuristics for Task Assignment in Distributed Systems*, in Proceedings of 1992 International Conference on Parallel and Distributed Systems, December 16-18, 1992, Hsinchu, Taiwan, pp.134-141.

[Tou93] Tout W.R., Pramanik S. - *A Distributed Load Balancing Scheme for Data Parallel Applications*, Proceedings of the 1993 Int. Conference on Parallel Processing, pp.II-213 - II-216.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

COMPLEXITY METRICS FOR DISTRIBUTED PROGRAMS

Monica CIACA*

Dedicated to Professor Sever Groze on his 65th anniversary

Received: March 15, 1995
AMS subject classification: 68Q15, 65Y05

REZUMAT. - Metrici ale complexităţii programelor distribuite. În lucrare se propun câteva metrici ale complexităţii programelor distribuite, metrici bazate pe diversele dependenţe de execuţie ce apar la nivelul unui program. Fiind bazate pe mai multe tipuri de dependenţă, metricile pot exprima complexitatea unui program distribuit din mai multe puncte de vedere.

1. Introduction

Metrics for measuring software complexity have many applications in software engineering activities including analysis, testing, debugging, and maintenance of programs, and management of project.

When we intend to measure some attributes of some entities, we must be able to capture information about the attributes and therefore we must have some representation and model of the entities such that the attributes can be explicitly described in the representation or the model. To measure the complexity of a concurrent program we must capture information about not only control structure and data flow in every process but also syncronization structure and interprocess communication among processes.

This paper presents some graph theoretical representations for distributed programs and defines some metrics for measuring complexity of distributed programs.

* *"Babeş-Bolyai" University, Faculty of Economics, 3400 Cluj-Napoca, Romania*

2. Definitions

A *nondeterministic parallel control flow net* (CFN) is a 10-tuple $(V,N,P_F,P_J,A_C,A_N,A_{PF},A_{PJ},s,t)$ where $(V,A_C,A_N,A_{PF},A_{PJ})$ is a simple arc-classified digraph such that $A_C \subseteq V \times V$, $A_N \subseteq N \times V$, $A_{PF} \subseteq P_F \times V$, $A_{PJ} \subseteq V \times P_J$, $N \subset V$ is a set of elements, called *nondeterministic selection vertices*, $P_F \subset V$ $(N \cap P_F = \Phi)$ is a set of elements, called *parallel execution fork vertices*, $P_J \subset V$ $(N \cap P_J = \Phi$, and $P_F \cap P_J = \Phi)$ is a set of elements, called *parallel execution join vertices*, $s \in V$ is a unique vertex, called *start vertex*, such that in-degree(s)=0, $t \in V$ is a unique vertex, called *termination vertex*, such that out-degree(t)=0 and $t \neq s$, and for any $v \in V$ $(v \neq s, v \neq t)$, there exists at least one path from s to v and at least one path from v to t. Any arc $(v_1,v_2) \in A_C$ is called *sequential control arc*, any arc $(v_1,v_2) \in A_N$ is called *nondeterministic selection arc*, and any arc $(v_1,v_2) \in A_{PF} \cup A_{PJ}$ is called *parallel execution arc*.

A usual (deterministic and sequential) control-flow graph can be regarded as a special case of nondeterministic parallel control-flow nets such that N, P_F, P_J, A_N, A_{PF}, and A_{PJ} are the empty set.

A *nondeterministic parallel definition-use net* (DUN) is a 7-tuple $(N_C, \Sigma_V, D, U, \Sigma_C, S, R)$, where $N_C=(V,N,P_F,P_J,A_C,A_N,A_{PF},A_{PJ},s, t)$ is a control flow-net, Σ_V is a finite set of symbols, called *variables*, $D:V \to P(\Sigma_V)$ and $P:V \to P(\Sigma_V)$ are two partial functions from V to the power set of Σ_V, Σ_C is a finite set of symbols, called *channels*, and $S:V \to \Sigma_C$ and $S:V \to \Sigma_C$ are two partial functions from V to Σ_C.

A DUN can be regarded as a CFN with the information concerning definitions and uses of variables and communication channels. A usual (deterministic and sequential) definition-use graph can be regarded as a special case of nondeterministic parallel definition-use nets such that $N,P_F,P_J,A_C,A_N,A_{PF},A_{PJ},\Sigma_C,S$ and R are the empty set.

The above definitions of CFN and DUN are graph-theoretical, and therefore, they are independent of any programming language.

3. Process dependence net

Program dependences are dependence relationships holding between statements in a program [Ferrante91]. Based on the DUN of a distributed program, we can formally define five kinds of primary program dependences: *control dependence, data dependence, selection dependence, synchronization dependence, and communication dependence* [Cheng92], [Cheng93].

Informally, a statement u is directly control-dependent on the control predicate v of a conditional branch statement (e.g., an if statement or while statement) if whether u is executed or not is directly determinated by the evaluation result of v; a statement u is directly data-dependent on a statement v if the value of a variable computed at v has a direct influence on the value of a variable computed at u; a statement u is directly selection-dependent on a nondeterministic selection statement v if whether u is executed or not is directly determined by the selection result of v; a statement u is directly synchronization-dependent on another statement v if the start and/or termination of execution of v directly determines whether or not the execution of u starts and/or terminates; a statement u in a process is directly communication-dependent on another statement v in another process if the value of a variable computed at v has a direct influence on the value of a variable computed at u by an interprocess communication.

If we represent all five kinds of primary program dependences in a distributed program within an arc-classified digraph such that each type of arcs represents a kind of primary

program dependences, then we can obtain an explicit dependence-based representation of the program. Such a representation is called process dependence net [Cheng92], [Cheng93].

The *process dependence net* (PDN) of a program is an arc-classified digraph **(V,Con,Sel,Syn,Com)** where **V** is the vertex set of the CFN of the program, **Con** is the set of control dependence arcs such that any **(u,v)∈Con** if and only if u is directly weakly control-dependent on v, **Sel** is the set of selection dependence arcs such that any **(u,v)∈Sel** if and only if u is directly selection-dependent on v, **Dat** is the set of data dependence arcs such that any **(u,v)∈Dat** if and only if u is directly data-dependent on v, **Syn** is the set of synchronization dependence arcs such that any **(u,v)∈Syn** if and only if u is directly synchronization-dependent on v and **Com** is the set of communication dependence arcs such that any **(u,v)∈Com** if u is directly communication-dependent on v.

4. Dependence-based complexity metrics

We will further use the following notations of relational algebra:

$\mathbf{R}^+$: the transitive closure of binary relation **R**.

$\sigma_{[1]=v}(\mathbf{R})$: the selection of binary relation **R** such that $\sigma_{[1]=v}(\mathbf{R})=\{(v_1,v_2)|(v_1,v_2)\in\mathbf{R} \text{ and } v_1=v\}$.

$\sigma_{[1]\in S}(\mathbf{R})$: the selection of binary relation **R** such that $\sigma_{[1]\in S}(\mathbf{R})=\{(v_1,v_2)|(v_1,v_2)\in\mathbf{R} \text{ and } v_1\in S\}$.

$\sigma_{[2]\in S}(\mathbf{R})$: the selection of binary relation **R** such that $\sigma_{[2]\in S}(\mathbf{R})=\{(v_1,v_2)|(v_1,v_2)\in\mathbf{R} \text{ and } v_2\in S\}$.

Based on the **PDN=(V,Con,Sel,Dat,Com)** of a distributed program, we can define the following dependence-metrics for measuring the complexity of the program, where $|A|$ is the cardinality of set A, $D_p\in\{\mathbf{Con,Sel,Dat,Syn,Com}\}$, $D_U=\mathbf{Con}\cup\mathbf{Sel}\cup\mathbf{Dat}\cup\mathbf{Syn}\cup\mathbf{Com}$, and P is the set of all statements of a process p:

$|D_P|/|D_U|$ is the proportion of a special primary program dependence to all primary program dependences in a program, and therefore, it can be used to measure the degree of concurrency of the program for a special viewpoint. For example, the larger are $|Con|/|D_U|$ and $|Dat|/|D_U|$ of a program, the less concurrent is the program. The larger is $|Sel|/|D_U|$ of a program, the more nondeterministic is the program. The larger are $|Syn|/|D_U|$ and $|Com|/|D_U|$ of a program, the more concurrent is the program.

$|Sel \cup Syn \cup Com|/|D_U|$ is the proportion of those primary program dependences concerning concurrency to all primary program dependences in a program, and therefore, it can be used to measure the degree of concurrency of the program from a general viewpoint.

$|D_P^+|/|D_U^+|$ is a metric different from $|D_P|/|D_U|$ in that $|D_P|/|D_U|$ only concerns direct dependences while $|D_P^+|/|D_U^+|$ concerns not only direct dependences but also indirect dependences.

$|(Sel \cup Syn \cup Com)^+|/|D_U^+|$: the difference between this metric and $|Sel \cup Syn \cup Com|/|D_U|$ is similar to the difference between $|D_P|/|D_U|$ and $|D_P^+|/|D_U^+|$.

$max/min\{|\sigma_{[i]\sim v}(D_P)|v \in V\}/|V|$ expresses the proportions of the maximal/minimal number of statements on which a statement is directly control, data, nondeterministic selection, synchronization, or communication dependent, respectively to the total number of statements in a program. The larger is the metric of a program, the more complex is the program. In particular, the larger is $max/min\{|\sigma_{[i]\sim v}(D_{CP})|v \in V\}/|V|$, where $D_{PC} \in \{Sel, Syn,$

Com}, of a program, the more complex is the concurrency in the program.

max/min$\{|\sigma_{[i]=v}(D_U)|v\in V\}/|V|$ expresses the proportions of the maximal/minimal number of statements on which a statement is somehow directly dependent, to the total number of statements in a program.

max/min$\{|\sigma_{[i]=v}(D_P^+)|v\in V\}/|V|$ expresses the proportions of the maximal/minimal number of statements on which a statement is directly and indirectly control, data, nondeterministic selection, synchronization, or communication dependent, respectively, to the total number of statements in a program.

max/min$\{|\sigma_{[i]=v}(D_U^+)|v\in V\}/|V|$ expresses the proportions of the maximal/minimal number of statements, on which a statement is somehow directly and indirectly dependent, to the total number of statements in a program.

The following metrics can be used to measure the degree of interaction among processes in a program. The larger is the maximal/minimal value of such a metric of a program, the more complex is the concurrency in the program:

$|\sigma_{[i]\in P}(Syn\cup Com)|/|V|$ is the proportion of the number of statements of other processes on which a process is directly dependent, to the total number of statements in a program.

$|\sigma_{[1]\in P}(Syn \cup Com)^+|/|V|$ is the proportion of the number of statements of other processes, which a process is directly and indirectly dependent on, to the total number of statements in a program.

$|\sigma_{[1]\in P1}(Syn \cup Com) \cap \sigma_{[2]\in P2}(Syn \cup Com)|/|V|$ is the proportion of the number of statements such that two processes directly depends on each other, to the total number of statements in a program.

$|\sigma_{[1]\in P1}(Syn \cup Com)^+ \cap \sigma_{[2]\in P2}(Syn \cup Com)^+|/|V|$ is the proportion of the number of statements such that two processes directly and indirectly depends on each other, to the total number of statements in a program.

Some of the above dependence-based complexity metrics can be used to measure the concurrency complexity of a distributed program in various aspects and some of them can be used to measure overall complexity of the program. Some other complexity metrics similar to the above metrics can also be considered. For a further discussion we refer the reader to [Conte86] and [Fenton91].

REFERENCES

[Cheng92] J.Cheng - Task Dependence Net as a Representation for Concurrent ADA Programs, in J. van Katwijk (ed.) "Ada: Moving towards 2000", *Lecture Notes in Computer Science*, Vol.603, Springer-Verlag, June 1992, pp.150-164.

[Cheng93] J.Cheng - Slicing Concurrent Programs, *Proc 1st International Workshop on Automated and Algorithmic Debugging*, May 1993.

[Conte86] S.D.Conte, H.E.Dunsmore and V.Y.Shen - *Software Engineering Metrics and Models*, Benjamin/Cummings, 1986, 396 p.

[Fenton91] N.E.Fenton - *Software Metrics: A Rigorous Approach*, Chapman&Hall, 1991, 337p.

[Ferrante87] J. Ferrante, K.J. Ottenstein and J.D. Warren - The Program Dependence Graph and its use in Optimization, *ACM TOPLAS*, Vol.9, No.3, July 1987, pp.319-349.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

DIVIDED DIFFERENCES AND CONVEX FUNCTIONS OF HIGHER ORDER ON NETWORKS

Maria-Eugenia IACOB*

Dedicated to Professor Sever Groze on his 65th anniversary

Received: November 16, 1994
AMS subject classification: 52-02

REZUMAT. - Diferenţe divizate şi funcţii convexe de ordin superior în reţele. În acest articol vom introduce diferenţele divizate pe n puncte, pentru orice funcţie reală, definită pe o submulţime conexă a unei reţele. Este expusă o teoremă de reprezentare a acestor diferenţe divizate generalizate. Ultima secţiune este dedicată funcţiilor convexe de ordin n, definite pe reţele şi studiului unor proprietăţi ale acestui tip de funcţii.

Abstract. For any real function f defined on a connected subset of an oriented network, we are interested to find a way to introduce divided differences on n points. We give a reprezentation theorem for the generalized divided differences and some properties resulting from this theorem.

Next we introduce the concept of convex real function of order n, defined on network. Some properties of these functions will be studied in the last part of this paper.

1. Notations and definitions

First we introduce the concept of network (see [1], [2], [3]). We consider a directed connected graph G = (W,A) without loops. To each vertex $i \in W = \{1,...,n\}$ we associate a point $v_i \in \mathbb{R}^3$. Thus yields a finite subset $V=\{v_1,...,v_n\}$ of $\mathbb{R}^3$, called the vertex set of the

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

network. We also associate to each arc $(i,j)\in A$ a rectifiable arc $[v_i,v_j]\subset\mathbb{R}^3$, called arc of the network, which has the orientation from v_i to v_j. Let assume that $[v_j,v_i]$ has the positive length e_{ij} and denote by U the set of all arcs. We define the network $N=(V,U)$ by the union

$$N = \bigcup_{(i,j)\in A} [v_i,v_j].$$

It is obvious that N is a geometric image of G, which follows naturally from an embedding of G in $\mathbb{R}^3$.

Suppose that for each $[v_i,v_j]$ in U there exists a continuous one-one mapping $T_{ij}:[0,1]\to[v_i,v_j]$ with $T_{ij}(0) = v_i$, $T_{ij}(1) = v_j$ and $T_{ij}([0,1]) = [v_i,v_j] \subset \mathbb{R}^3$.

Let Q_{ij} be the inverse of T_{ij}. To each point x from $[v_i,v_j]$ corresponds a unique point $Q_{ij}(x)$ in $[0,1]$.

Any connected and closed subset of an arc $[v_i,v_j]$, bounded by two points x and y of $[v_i,v_j]$ and having the same orientation as $[v_i,v_j]$, is called a closed subarc and is denoted by $[x,y]$. If one or both of x, y miss we say that the subarc is open in x (or in y) or is open and we denote this by $[x,y)$ or $(x,y]$ or (x,y), respectively.

Using Q_{ij}, it is possible to compute the length of $[x,y]$ as

$$e([x,y]) = |Q_{ij}(x) - Q_{ij}(y)|e_{ij}.$$

Particulary we have $e([v_i,v_j]) = e_{ij}$, $e([v_i,x]) = Q_{ij}(x)e_{ij}$ and $e([x,v_j]) = (1 - Q_{ij}(x))e_{ij}$.

Definition 1.1 A chain L(x,y) linking two points x and y in N is a sequence of arcs and at most two subarcs at extremities. The length of a chain is the lengths sum of all its component arcs and subarcs. If the chain L(x,y) contains only distinct vertices then we call it elementary.

Definition 1.2. A route D(x,y) starting from x and ending in y ($x,y \in N$) is a chain, which has the same orientation for all the component arcs and subarcs. This also is the route

orientation.

Let $L^*(x,y)$ be one of the shortest chains and $D^*(x,y)$ one of the shortest routes between the points x,y in N. We define in N a distance as follows:

$d(x,y) = e(L^*(x,y))$ for any x,y in N.

It is obvious that d is a metric on the oriented network N.

2. Divided differences

Our purpose in this section is to extend in a natural way the divided differences on n points of real functions (see [7], [8], [4]) for the functions defined on networks.

In order to define the divided differences of a function f on n points we will consider the notion of metric segment.

Definition 2.1 The metric segment between two different points $x,y \in N$ is the following set:

$$\langle x,y\rangle = \{z \in N \mid d(x,z) + d(z,y) = d(x,y)\}.$$

Remark. Another way of stating definition 2.1, in geometric language, is to say that the metric segment $\langle x,y\rangle$ coincides with the union of all the shortest chains between x and y.

So, it is easy to see why the above remark leds us to the following notion.

Definition 2.2 The metric oriented segment between $x,y \in N$ is the set:

$$\langle x,y\rangle_o = \{z \in N \mid z \in D^*(x,y) \text{ and } d(x,z)+d(z,y)=d(x,y)\},$$

where $D^*(x,y)$ denote a shortest route linking the points $x,y \in N$.

Remarks. 1. Following the definition 2.2 the oriented metric segment is the union of all shortest routes from x to y which have the length d(x,y). Thus we can see that $\langle x,y\rangle_o$

could be empty.

2. It is obvious that $\langle x,y\rangle_o \subset \langle x,y\rangle$.

3. $\langle x,y\rangle_o \neq \langle y,x\rangle_o$.

Let $x,y \in N$ be two distinct points such that $\langle x,y\rangle_o \neq \emptyset$ and consider a shortest route $D^*(x,y) \subset \langle x,y\rangle_o$. We introduce the function $d_{xy}: D^*(x,y) \times D^*(x,y) \to \mathbb{R}$,

$$d_{xy}(z_1,z_2) = \begin{cases} -d(z_1,z_2), \text{ if from } z_1 \text{ we can reach} \\ \qquad\qquad z_2 \text{ on } D^*(x,y) \text{ following} \\ \qquad\qquad \text{the route orientation} \\ d(z_1,z_2), \text{ if contrary} \end{cases}$$

Here are some elementary properties of d_{xy}.

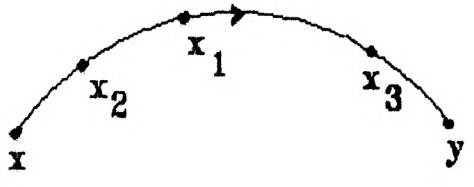
Lemma 2.1 a. $d_{xy}(z_1,z_2) = -d_{xy}(z_2,z_1)$, $\forall z_1,z_2 \in D^*(x,y)$.

b. $\forall x_1,x_2,x_3 \in D^*(x,y)$ are loc $d_{xy}(x_1,x_2)+d_{xy}(x_2,x_3)=d_{xy}(x_1,x_3)$.

Proof.

a. This equality is a direct consequence of d_{xy}'s definition.

b. Let us consider the points x,y,x_1,x_2,x_3 like in the following figure.



Then $d_{xy}(x_1,x_2) + d_{xy}(x_2,x_3) = d(x_1,x_2) - d(x_2,x_3) =$

$= -d(x_1,x_3) = d_{xy}(x_1,x_3)$.

The same reasoning applies to the case of any permutation of points x_1,x_2,x_3.■

Definition 2.3 Let $A\subset\mathbb{N}$ and consider the real function $f:A\to\mathbb{R}$. We choose the points $x_1,\dots,x_n\in A$ pairwise distinct and such that there exist $x,y\in A$ and a shortest route $D^*(x,y)\subset\langle x,y\rangle_o$ for which is fullfiled the condition $x_1,\dots,x_n\in D^*(x,y)$. We call divided differecence of function f on points $x_1,\dots,x_n$ the number:

$$[x_1,\dots,x_n;f]=\frac{[x_2,\dots,x_n;f]-[x_1,\dots,x_{n-1};f]}{d_{xy}(x_n,x_1)}$$

where for any $z,t\in D^*(x,y)\cap A, z\neq t$, we set

$$[z,t;f]=\frac{f(t)-f(z)}{d_{xy}(t,z)}.$$

The following divided differences reprezentation formula holds:

Theorem 2.1 Given $f:A\to\mathbb{R}$ ($A\subset\mathbb{N}$) and the points $x_1,\dots,x_n$ under the conditions stated in definion 2.3, then

(1) $$[x_1,\dots,x_n;f]=\sum_{k=1}^{n}\frac{f(x_k)}{p_k(x_1,\dots,x_n)},\ \text{where}$$

$$p_k(x_1,\dots,x_n)=d_{xy}(x_k,x_1)\dots d_{xy}(x_k,x_{k-1})d_{xy}(x_k,x_{k+1})\dots d_{xy}(x_k,x_n).$$

Proof. We shall prove (1) by induction. For n=3 we have:

$$[x_1,x_2,x_3;f]=\frac{[x_2,x_3;f]-[x_1,x_2;f]}{d_{xy}(x_3,x_1)}=\frac{\dfrac{f(x_3)-f(x_2)}{d_{xy}(x_3,x_2)}-\dfrac{f(x_2)-f(x_1)}{d_{xy}(x_2,x_1)}}{d_{xy}(x_3,x_1)}=$$

$$=\frac{f(x_1)}{d_{xy}(x_2,x_1)d_{xy}(x_3,x_1)}-\frac{f(x_2)}{d_{xy}(x_3,x_1)}\left(\frac{1}{d_{xy}(x_3,x_2)}+\frac{1}{d_{xy}(x_2,x_1)}\right)+$$

$$+\frac{f(x_3)}{d_{xy}(x_3,x_1)d_{xy}(x_3,x_2)}$$

Using lemma 2.1 it follows

$$[x_1,x_2,x_3;f]=\sum_{k=1}^{3}\frac{f(x_k)}{p_k(x_1,x_2,x_3)}.$$

Assume (1) is true and prove the property holds for n+1 points, that is:

$$[x_1,\dots,x_{n+1};f] = \sum_{k=1}^{n+1} \frac{f(x_k)}{p_k(x_1,\dots,x_{n+1})}.$$

According to definition 2.3 and using the induction hypothesis we have:

$$[x_1,\dots,x_{n+1};f] = \frac{1}{d_{xy}(x_{n+1},x_1)}\left[\sum_{k=1}^{n} \frac{f(x_{k+1})}{p_{k+1}(x_2,\dots,x_{n+1})} - \sum_{k=1}^{n} \frac{f(x_k)}{p_k(x_1,\dots,x_n)}\right] =$$

$$= \frac{1}{d_{xy}(x_{n+1},x_1)}\left[- \frac{f(x_1)}{d_{xy}(x_1,x_2)\dots d_{xy}(x_1,x_n)} + \right.$$

$$+ f(x_2)\left(\frac{1}{d_{xy}(x_2,x_3)\dots d_{xy}(x_2,x_{n+1})} - \right.$$

$$\left. - \frac{1}{d_{xy}(x_2,x_1)d_{xy}(x_2,x_3)\dots d_{xy}(x_2,x_n)}\right) + \dots +$$

$$+f(x_n)\left(\frac{1}{d_{xy}(x_n,x_2)\dots d_{xy}(x_n,x_{n-1})d_{xy}(x_n,x_{n+1})} - \right.$$

$$\left. - \frac{1}{d_{xy}(x_n,x_1)\dots d_{xy}(x_n,x_{n-1})}\right) +$$

$$\left. + \frac{f(x_{n+1})}{d_{xy}(x_{n+1},x_1)\dots d_{xy}(x_{n+1},x_n)}\right].$$

Taking in account lemma 2.1 under the transcription

$$d_{xy}(x_1,x_k)+d_{xy}(x_k,x_{n+1})=d_{xy}(x_1,x_{n+1}),k=2,\dots,n \text{ and}$$

$$d_{xy}(x_k,x_j) = - d_{xy}(x_j,x_k),k=1,..,n+1,j=1,\dots,n+1,$$

we conclude:

$$[x_1,\dots,x_{n+1};f] = \frac{1}{d_{xy}(x_{n+1},x_1)}\left[-\frac{f(x_1)}{d_{xy}(x_1,x_2)\dots d_{xy}(x_1,x_n)} - \right.$$

$$-\frac{f(x_2)d_{xy}(x_1,x_{n+1})}{d_{xy}(x_2,x_1)\dots d_{xy}(x_2,x_{n+1})} - \dots -$$

$$\left.-\frac{f(x_n)d_{xy}(x_1,x_{n+1})}{d_{xy}(x_n,x_1)\dots d_{xy}(x_n,x_{n-1})d_{xy}(x_n,x_{n+1})}+\frac{f(x_{n+1})}{d_{xy}(x_{n+1},x_2)\dots d_{xy}(x_{n+1},x_n)}\right] =$$

$$= \sum_{k=1}^{n+1}\frac{f(x_k)}{p_k(x_1,\dots,x_{n+1})}. \blacksquare$$

Theorem 2.1 is an analogue for networks of a similar result concerning the usual divided differences ([7], [8], [4]). A immediate consequence of this characterisation theorem is:

Theorem 2.2 If f, g are real function defined on $A\subset N$, $\alpha\in\mathbb{R}$ and $x_1,\dots,x_n\in N$ satisfy the conditions stated in definition 2.3, then:

$$[x_1,\dots,x_n;f+g] = [x_1,\dots,x_n;f] + [x_1,\dots,x_n;g]$$

$$[x_1,\dots,x_n;\alpha f] = \alpha[x_1,\dots,x_n;f].$$

Proof. The above relations follow directly from (1).■

3. Functions of order n on networks.

The last part of this paper is devoted to define and study the functions of higher order.

Definition 3.1 If $x_1,\dots,x_n\in N$ are n points (n>1) pairwise disctinct, we call this points a metric sequence if the following conditions are fullfiled:

a. There exist $x,y\in N$ and $D^*(x,y)\subset\langle x,y\rangle_o\neq\emptyset$ such that $x_1,\dots,x_n\in D^*(x,y)$.

b. $d_{xy}(x_1,x_2) < 0$.

c. $\sum_{k=1}^{n-1} d(x_k,x_{k+1}) = d(x_1,x_n)$.

d. $d(x_{k-1},x_k)+d(x_k,x_{k+1})=d(x_{k-1},x_{k+1})$, $k=2,\dots,n-1$.

Remark. One can easily seen that any subsequence of $\{x_1,\dots,x_n\}$ is also a metric sequence.

Theorem 3.1 If f is a real valued function defined ACN and $x_1,\dots,x_n\in A$ is a metric sequence, then

$$(2)\quad [x_1,\dots,x_n;f]=\sum_{k=1}^{n}\frac{(-1)^{n-k}f(x_k)}{d(x_k,x_1)\dots d(x_k,x_{k-1})d(x_k,x_{k+1})\dots d(x_k,x_n)}$$

Proof. Since $x_1,\dots,x_n$ is a metric sequence it follows

$$(3)\qquad d_{xy}(x_k,x_j)=\begin{cases}-d(x_k,x_j), & \text{dacă } j>k\\ d(x_k,x_j), & \text{dacă } j<k\end{cases}$$

Using (3) in (1) we obtain (2).

Definition 3.2 A real valued function f defined on ACN is called convex, notconcave, polynomial, notconvex, concave of order n on A if for any metric sequence $x_1,\dots,x_{n+2}\in A$ the following inequalities holds respectively:

$$[x_1,\dots,x_{n+2};f]>,\ \geq,\ =,\ \leq,\ <0.$$

All these functions are of order n.

Remark. If we take n=1 for notconcavity in definition 3.2 we recover the usual d-convex functions in metric spaces ([10]).

Definition 3.3 A real valued function $f:A\to\mathbb{R}$ is d-convex on A ACN if

$$(4)\qquad f(z)\leq\frac{d(z,y)}{d(x,y)}f(x)+\frac{d(x,z)}{d(x,y)}f(y),$$

for any points x,y,z in A such that $z\in\langle x,y\rangle_o$ (which is the same with saying that x,z,y is metric sequence).

(4) is equivalent with

$$\frac{f(x)}{d(x,y)d(x,z)} - \frac{f(z)}{d(x,z)d(y,z)} + \frac{f(y)}{d(x,y)d(y,z)} \geq 0.$$

Using (2) the above relation becomes:

$$[x,z,y;f] \geq 0, \ \forall x,z,y \in A$$

where x,z,y is a metric sequence. This corresponds to definition 3.2 in the case of first order notconcave functions.

Definition 3.4 A circuit in a oriented network N is a route D(x,y) with x=y. In what follows C allways denote a circuit.

The main result of this section is:

Theorem 3.2 If C is a elementary circuit, then any real function $f:C\to\mathbb{R}$ of order n is polynomial.

Proof. We shall prove that any notconcave function is polynomial on C. The same reasonning applies to notconvex functions. It is obvious that if the above statement is true then there exist no convex or concave function of order n on C.

If $f:C\to\mathbb{R}$ is notconcave, then for any metric sequence $x_1, \ldots, x_{n+2} \in C$ we have

(5) $$[x_1,\ldots,x_{n+2};f]\geq 0.$$

Consider a metric sequence $x_1, \ldots, x_{n+2} \in C$. It is easy to see that there exists m>n and the points $x_{n+3}, \ldots, x_m$ satifying the conditions:

1. The numbering of this points is made such that for any $k\in\{1,\ldots,m\}$, from x_k we can reach x_{k+1} following the orientation of C. We assume $x_{m+1}=x_1$.

2. The sets $\{x_1,\ldots,x_{n+2}\},\{x_2,\ldots,x_{n+3}\},\ldots,\{x_m,x_1,\ldots,x_{n+1}\}$

are metric sequences.

Using definition 2.3 and (5) we obtain:

$$(6) \qquad [x_i,\dots,x_{i+n+1};f] = \frac{[x_{i+1},\dots,x_{i+n+1};f]-[x_i,\dots,x_{i+n};f]}{d(x_{i+n+1},x_i)} \geq 0,$$

for each $i\in\{1,\dots,m\}$ and under the assumption that

$$x_{i+n+1} = \begin{cases} x_{i+n+1}, & \text{if } i+n+1\leq m \\ x_{i+n+1-m}, & \text{if } i+n+1>m \end{cases}$$

It is obvious that (6) implies:

$$[x_{i+1},\dots,x_{i+n+1};f] \geq [x_i,\dots,x_{i+n};f], \ \forall i\in\{1,\dots,m\}.$$

Thus we can write the following sequence of inequalities:

$[x_1,\dots,x_{n+1};f] \leq [x_2,\dots,x_{n+2};f] \leq \dots \leq [x_m,x_1,\dots,x_n;f] \leq$

$\leq [x_1,\dots,x_{n+1};f]$. This clearly forces the equality:

$$(7) \quad [x_1,\dots,x_{n+1};f] = [x_2,\dots,x_{n+2};f] = \dots = [x_m,x_1,\dots,x_n;f].$$

Using (7) in (6) yields $[x_1,\dots,x_{n+2};f]=0$. Since the metric sequence was arbitrarily choosen we can conclude that f is polinomial of order n on C.■

It is ovious that

Corollary 3.1 A n order real function is polinomial on any union of elementary adjacent circuits.

Corollary 3.2 Any real valued function f, defined on a set $A\subset N$ which contains a elementary circuit C cannot be convex or concave of order n, whatever the natural number n is.

For usual d-convex functions we sate now a stronger result. The proof is adapted from [9], p. 127.

Theorem 3.3 Any real valued and d-convex function, defined on a elementary circuit

C is constant.

Proof. We consider the d-convex function $f:C\to\mathbb{R}$, where $C\subset N$ is a elementary circuit. We want to prove that $f(x)=f(y)$ holds for any pair $x,y\in C$. It is easy to see that there exist the points $z_1, \dots, z_n\in C$ ($n\geq 5$) satifying the conditions:

1. $z_i\in\langle z_{i-1},z_{i+1}\rangle_o$, $i=2,\dots,n$, where $z_{n+1}=z_1$,

2. $z_1=x$ and $\exists k\in\{2,\dots,n\}$ such that $z_k=y$.

Assume f reach the maximum value on the set $\{z_1,\dots,z_n\}$ in z_p. Since f is d-convex we have:

$$f(z_p)\leq \frac{d(z_{p+1},z_p)}{d(z_{p-1},z_{p+1})}f(z_{p-1}) + \frac{d(z_{p-1},z_p)}{d(z_{p-1},z_{p+1})}f(z_{p+1})\leq$$

$$\leq \frac{d(z_{p+1},z_p)}{d(z_{p-1},z_{p+1})}f(z_p) + \frac{d(z_{p-1},z_p)}{d(z_{p-1},z_{p+1})}f(z_p) = f(z_p),$$

which leds us to $f(z_p)=f(z_{p-1})=f(z_{p+1})$. Repeated application of this reasoning enable us to write $f(z_1)=\dots=f(z_n)$, and thus $f(x)=f(y)$.

Since x and y was arbitrarily chosen we conclude that f is constant on C.■

Definition 3.5 We call the elementary circuits $C_1,C_2\subset N$ adjacent if $C_1\cap C_2\neq\emptyset$.

Corollary 3.3 Any d-convex function defined on the connected set $A\subset N$ is constant on the union of all adjacent circuits in A.

Proof. This corollary is the direct consequence of theorem 3.3 and definition 3.5.■

We mention now some elementary properties of functions of order n.

Theorem 3.4

1. Given the real number $\alpha>0$ and two real functions convex (notconcave, polynomial,

notconvex, concave) of order n f,g defined on $A \subset N$, then f+g and αf are also convex (notconcave, polynomial, notconvex ,concave) of order n.

2. The limit of a punctualy convergent sequence of convex (or notconcave) functions of order n is notconcave of order n.

3. The limit of a punctualy convergent sequence of concave (or notconvex) functions of order n is notconvex of order n.

4. The limit of a punctualy convergent sequence of polynomial functions of order n is also polynomial of order n.

Proof.

1. This follows from theorem 2.2.

2. Let us consider a metric sequence $x_1,\dots,x_{n+2} \in N$ and $f_i: N \to \mathbb{R}$, convex (or notconcave) of order n, for each $i \in \mathbb{N}$. If $f: N \to \mathbb{R}$, $f(z) = \lim_{i \to \infty} f_i(z)$ then

$$[x_1,\dots,x_{n+2};f] =$$

$$= \sum_{k=1}^{n+2} \frac{(-1)^{n+2-k} f(x_k)}{d(x_k,x_1)\dots d(x_k,x_{k-1}) d(x_k,x_{k+1})\dots d(x_k,x_{n+2})} =$$

$$= \sum_{k=1}^{n+2} \frac{(-1)^{n+2-k} \lim_{i \to \infty} f_i(x_k)}{d(x_k,x_1)\dots d(x_k,x_{k-1}) d(x_k,x_{k+1})\dots d(x_k,x_{n+2})} =$$

$$= \lim_{i \to \infty} \sum_{k=1}^{n+2} \frac{(-1)^{n+2-k} f_i(x_k)}{d(x_k,x_1)\dots d(x_k,x_{k-1}) d(x_k,x_{k+1})\dots d(x_k,x_{n+2})} =$$

$$= \lim_{i \to \infty} [x_1,\dots,x_{n+2};f_i] \geq 0.$$

For 3. and 4. one can use a proof similar with that made for 2. ■

The technique we use here to introduce divided differences and function of higher order allows us to make other natural extensions to networks of some types of generalized

convex function, for example cvasiconvex function of order n ([5], [6]).

REFERENCES

[1] Dearing P.M., Francis R.L., Lowe T.J., Convex location problems on tree networks, Oper. Res., 24(1976), 628-634.

[2] Hooker J., Nonlinear Network Location Models,Ph.D. Thesis, Univ. of Microfilms Int., Ann Arbor, 1984.

[3] Labbé, M., Essay in network location theory, Cahiers du Centre d'étude et de Recherche Opérationelles, vol. 27, nr 1-2, 1985, pp 7-130.

[4] Popoviciu, E., Teoreme de medie din analiza matematică şi legătura lor cu teoria interpolării, Dacia, Cluj, 1972.

[5] Popoviciu E., Sur une allure de quasiconvexite d'ordre superieur, Rev. d'Anal. Num. Theor.l'Approx., 1982, 11(1-2), 129-137.

[6] Popoviciu, E., Sur quelques proprietés des fonctions quasi-convèxes, Preprint nr. 2, 1983, Itinerant seminar on functional equations approximation and convexity, Cluj-Napoca.

[7] Popoviciu, T., Introduction à la théorie des différences divisées, Bull. Math. de la Soc. Roumaine de Science, 43, nr. 1-2 , 1941.

[8] Popoviciu,T., Les fonctions convèxes, Actualités scientifiques et industrielles, 992, XVII, Paris, 1945.

[9] Soltan, V.P., Introducere în teoria axiomatică a convexităţii, Ştiinţa, Chişinău, 1984.

[10] Soltan, V.P., Some properties of d-convex Functions, I şi II, Amer. Math. Soc. Transl., (2), vol 137, 1987.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

DETECTING DEADLOCKS IN MULTITHREADED APPLICATIONS

Simona Daniela IURIAN*

Dedicated to Professor Sever Groza on his 65th anniversary

Received: February 8, 1995
AMS subject classification: 68Q05, 68Q10, 68Q60, 68Q90

REZUMAT. - Detectarea impasului în aplicaţii cu mai multe fire de execuţie. Este prezentată o modalitate de descriere a aplicaţiilor cu fire de execuţie multiple, bazată pe reţelele Petri. Folosind acest model şi rezultate din teoria reţelelor Petri se dă un algoritm de detectare a impasului într-o astfel de aplicaţie.

A threads package

We introduce a specification of a threads package, which will be referred further in this paper for describing and analysing a multithreaded application. This package is an extension of the 'C' language and is suggested by the threads library from Windows NT (see [3]). So, for developing multithreaded application we will use 'C' enriched with few data types and functions, which are listed bellow.

The new data types and constants are:

typedef BYTE *PThreadID;	A handle for the thread
typedef FARPROC *PThreadFunction;	The function which contains the code of a thread
typedef int TCriticalRegion	Data type for a condition variable
typedef void far * TEvent	Data type for an event

The functions which are interesting for our purpose are presented bellow:

· PThreadId ThreadCreate(TThreadFunction ThreadFn)

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

The call of this function try to create a new thread. The returned value is a handle for the new thread and could be used in the future to refer it. The parameter ThreadFn indicates the code which will be executed by the new thread.

- void EnterCriticalRegion(TCriticalRegion RC)

  A thread has to call this function if it needs to enter the critical region RC (see [2]). If the critical region RC is not available, the thread will wait for the other thread which executes RC to release it.

- void ReleaseCriticalRegion(TCriticalRegion RC)

  A thread call this function if it finished the execution of the critical region RC. The critical section RC become available to another thread.

Also, it can be defined other function which permits to stop the execution of a thread, to resume the execution of a stopped thread and for synchronize multiple threads by waiting for the ocurrence of an event.

Writing a multithreaded application as a Petri Net

In this section we will discuss the synchronisation mechanisms and the creation and termination primitives which were introduced until now in terms of Petri nets. The analogies will be used to write a multithreaded application as a Petri net. The resulting Petri nets can be used to analyse the properties of a multithreaded application.

An application with a single thread can be written as a Petri net by associating to any instruction a transition and introducing a place between any two consecutive transitions, an initial place before the first transition and, (if any) a final place after the last(s) transition(s). A place define the state of the application (in fact, of the thread) and a transition represent an action to be taken. At the beginning is marked just the initial place of the Petri net, with

a single token. At any moment, the marked place indicate the point where the execution of the thread is arrived.

The Petri nets can be used as an abstracting mechanism. Most of the instructions are irrelevant for our purpose. The only interesting operations are the threads synchronisation primitives and the control structures, if the different branches contains such synchronisation primitive or are creating new threads. So, we will ignore the uninteresting operations: the assignments and the calls of 'C' library functions. Varshavsky, in [5] presents a possibility to describe a sequential program as a Petri Net.

We will associate to a multithreaded application a Petri net in which the places may contain zero or a single token (this is a condition/event system, see [2] for details). Reisig, in [2], defines the notion of invariant in a Petri net. This is a vector I for which M*I=0, where M is the incidence matrix associate to the Petri net. It is shown that, for each case C of the Petri net and for each invariant I, the scalar product C*I is constant.

In order to write a multithreaded application as a Petri Net we need to define the configurations of transitions/places which corresponds to thread control primitives and have to be added to those described in [5].

ThreadCreate

In terms of Petri Nets, a thread can be viewed as a linear sequence of places and transitions. Each thread has a start place. The start place of the main thread will be marked. The start place of the others threads will be initially not marked.

If a thread creates a new thread, it will be created a new execution sequence corresponding to the newly created thread. This situation is depicted in figure 1a. If a thread executes the function ThreadCreate, the first place of the new thread and the next place of the

old thread will become marked and further the two threads will be executed in parallel. It follows then, in a multithreaded application there could be more than one place marked at a moment.

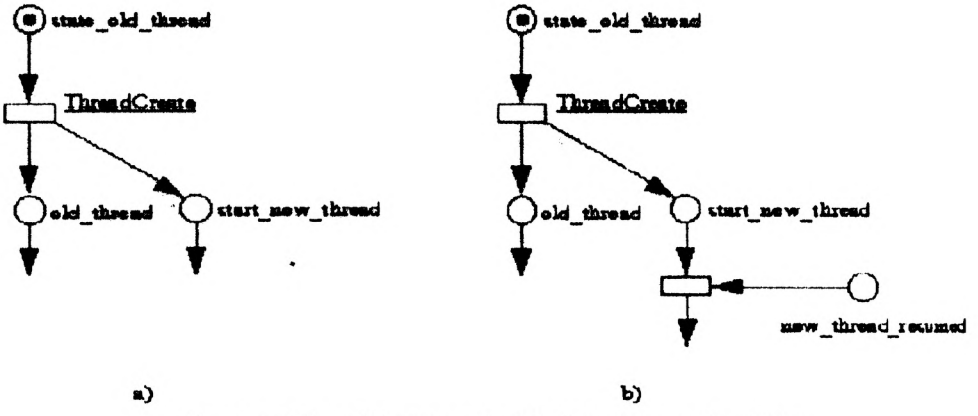


Figure 1. ThreadCreate and critical regions as Petri Nets

The critical region

If more than one thread needs to execute codes wh: h are mutually exclusive, they have to be synchronised. Since, the entering of such a code will begin with a call of EnterCriticalRegion, and the leaving of such a code will be marked by a call of ReleaseCriticalregion. The corresponding Petri net is depicted in figure 1b.

Detecting deadlocks in a multithreaded application

The deadlock is a very acute problem which can appear in a multithreaded application (see [4]). A deadlock is a state in which two or more threads wait each other to release a shared resource which may not be accesed simultaneously by the two threads. Because all threads participating to a deadlock are suspended and cannot release the shared resource, the whole application is blocked.

The next example describe a possible deadlock and will be used from now on.

```
TCriticalRegion rc1, rc2;

main()
{
        int Thread2ID=ThreadCreate(Thread2);
        while (1)
        {
                EnterCriticalRegion(rc1);
                EnterCriticalRegion(rc2);
                ReleaseCriticalRegion(rc2);
                ReleaseCriticalRegion(rc1);
                // do something;
        }
}
int Thread2()
{
        while (1)
        {
                EnterCriticalRegion(rc2);
                EnterCriticalRegion(rc1);
                ReleaseCriticalRegion(rc1);
                ReleaseCriticalRegion(rc2);
                // do something;
        }
}
```

It is obvious that there is a situation which represents a deadlock: when the main thread enter the critical region rc1, and the second thread enter the critical region rc2 at the same time. Then, the main thread wants to enter the critical region rc2, which is owned by the second thread. Also, the second thread waits for the releasing of the critical region rc1 which is blocked because of the main thread, and so on.

Generally, the deadlock is not as easy to detect. For detecting deadlocks there are two strategies: the posthumous way and the use of the invariants. The posthumous way consist of describing the application and executing it until it seems to be appeared a deadlock. After this, the problem is solved and the method is reitered.

In order to present the invariants method we will rewrite the previous example.

```
int Thread2()
{
        while (1)
        {
                EnterCriticalRegion(rc1);
                EnterCriticalRegion(rc2);
                ReleaseCriticalRegion(rc2);
                ReleaseCriticalRegion(rc1);
                // do something;
        }
}
```

Now, it is "obvious" that there could not appear a deadlock. For showing that we will enunciate a few evident propositions related to the flow control in the new application:

1. The critical region rc1 is either free or is accesed by Thread1, or is accesed by Thread2 (it follows from the definition of a critical region).
2. If the critical region rc1 is free, then Thread1 and Thread2 are executing the code from the beginning of the corresponding loop.
3. If the critical region rc1 is accesed by Thread1 (respectively Thread2), then Thread2 (respectively Thread1) is executing the code from the beginning of his while instruction, or is waiting for releasing the critical region rc1.
4. The critical region rc2 can be requested only after the critical region rc1 is accesed.
5. If the critical region rc2 is accesed by a thread, then this thread execute something between EnterCriticalRegion(rc2) and ReleaseCriticalRegion(rc2), and the other thread execute the code from the beginning of his while instruction or is waiting for the releasing of the critical region rc1.

These propositions are named invariants, because they are true at any moment of the application's execution. In a deadlock situation (from the definition), a thread A is suspended waiting the releasing of a critical region RC which is accessed by the thread B ($A \neq B$). The

thread B must be waiting for the releasing of the critical region RC' (RC'≠RC) by the thread A. But the two threads cannot be simultaneously in a critical region (it follows from the above propositions), so a deadlock cannot appear.

To show formally that a multithreaded application is deadlock-free we may use the next algorithm:

1. Writing the multithreaded application as a Petri net.
2. Computing the Petri net invariants.
3. Generating all possible cases for the Petri net.
4. For each generated case:
   - Verifying if the case can be obtained from the initial case
   - If so, verifying if it represents a deadlock situation
   - If so, trying if that case verify the invariants. If all invariants are verified, then it is possible to appear a deadlock situation. If there is at least an invariant which is not verified by the case, then the multithreaded application cannot lead to the localised deadlock.
5. If there is at least a case, representing a deadlock, which can be obtained from the initial case and is verifying the application invariants, then there is possible to appear a deadlock in the execution of the application. If there is not such a case, then the application cannot lead to a deadlock situation.

   For our example, the above steps are detailed further.

   The Petri net for our initial application is depicted in figure 2.

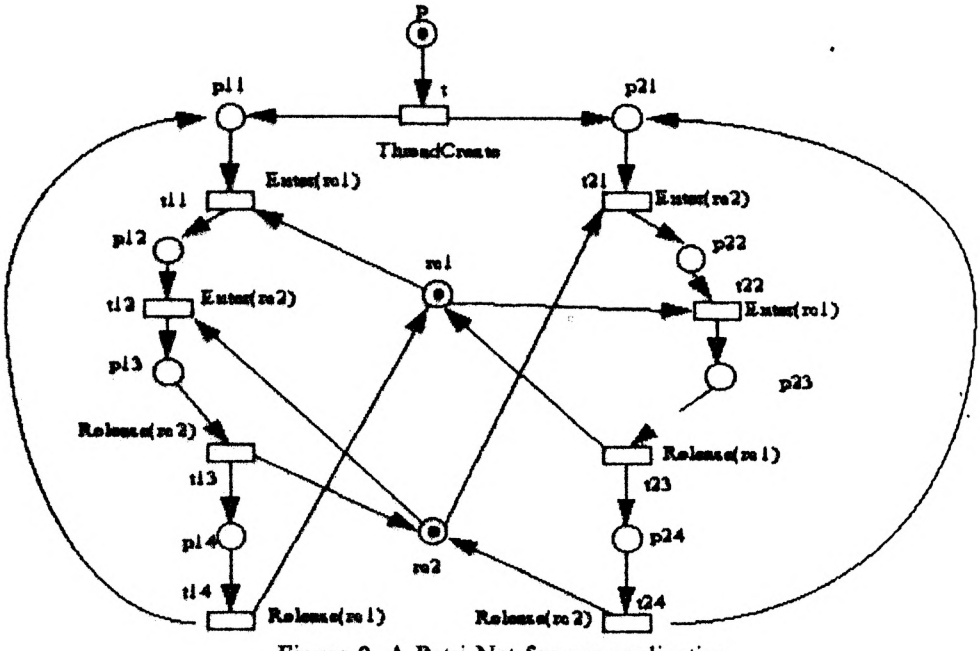


Figure 2. A Petri Net for our application

The incidence matrix (see [2]) M for this Petri net is the following:

	p	p11	p12	p13	p14	rc1	rc2	p21	p22	p23	p24
t	-1	1	0	0	0	0	0	1	0	0	0
t11	0	-1	1	0	0	0	-1	0	0	0	0
t12	0	0	-1	1	0	-1	0	0	0	0	0
t13	0	0	0	-1	1	1	0	0	0	0	0
t14	0	1	0	0	-1	0	1	0	0	0	0
t21	0	0	0	0	0	-1	0	-1	1	0	0
t22	0	0	0	0	0	0	-1	0	-1	1	0
t23	0	0	0	0	0	0	1	0	0	-1	1
t24	0	0	0	0	0	1	0	1	0	0	-1

The solutions I of the equation M*I=0 are the basis of the vector space that contains all invariants of the application (see [2]). In our case, the solutions are:

No.	p	p11	p12	p13	p14	rc1	rc2	p21	p22	p23	p23
1	1	1	1	1	1	0	0	0	0	0	0
2	1	0	0	-1	0	-1	0	1	0	0	0
3	-1	-1	0	0	0	0	1	0	0	1	0
4	1	1	0	1	0	1	-1	0	1	0	1

The 11 columns represents the 11 places. The meaning of an invariant is that the sum of the tokens from the corresponding places, multiplied by the number, is constant. In other words:

p+p11+p12+p13+p14 = constant

p+p21-p13-rc1 = constant

-p+rc2+p23-p11 = constant

p+p11+p13+rc1-rc2+p22+p24 = constant

By taking into account the initial case for the Petri net (that shown in figure 3) we can compute the real value of the constants from the above relations. So, because in the initial case, p=1, rc1=1, rc2=1 and the others places have no tokens (for those, the value is 0) we obtain the following relations:

p+p11+p12+p13+p14 = 1

p+p21-p13-rc1 = 0

-p+rc2+p23-p11 = 0

p+p11+p13+rc1-rc2+p22+p24 = 1

A case which represent a deadlock is <p12, p22>. For showing this we can construct the resource allocation graph (see [4]) and finding a cycle in this. For that case, it is very easy to see that the invariants are satisfied. Unfortunately, the verification of the invariants is a condition necessary but not sufficient for a case to be attained by starting from the initial case.

We can conclude that in our application it may occurs this deadlock.

REFERENCES

1. Krishnamurty, E.V., "Parallel Processing. Principles and Practice", Addison-Wesley Publishing Company, 1989
2. Reisig, W., "Petri Nets. An Introduction", Springer-Verlag, Berlin, 1985.
3. Richter, J., "Advanced Windows NT", Microsoft Press, 1994.
4. Tannenbaum, A., "Modern Operating Systems", Prentice-Hall, 1992
5. Varshavsky, V., "Self-Timed Control of Concurrent Processes", Kluwer Academic Publishers, 1990.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

A STUDY OF THE PROPERTIES OF THE FUZZY RELAXATION ALGORITHM

Horia F. POP*

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 1, 1995
AMS subject classification: 68P99

REZUMAT. - Un studiu asupra proprietăţilor Algoritmului de Relaxare Fuzzy. Una dintre problemele cele mai dificile ale instruirii supervizate este tratarea datelor neseparabile liniar. Problema a fost doar parţial rezolvată prin utilizarea algoritmilor de instruire nuanţată. Una din posibilităţile de abordare a problemei este generalizarea pentru cazul neseparabil a unor tehnici de instruire nuanţată care funcţionează bine în cazul separabil. In cele ce urmează vom studia proprietăţile Algoritmului de Relaxare Nuanţată [1,2]. Acesta permite o generalizare favorabilă datelor neseparabile liniar.

1. Introduction

In [1,2] it has been proposed a new training method that allows the use of fuzzy sets in order to develop the training. Based on this method a series of algorithms representing generalizations of some well known classical algorithms have been given.

There have also been proposed robust variants of the fuzzy training algorithm. These robust algorithms are capable of learning a training set consisting of two fuzzy linearly non-separable classes.

In this paper we study the fuzzy relaxation algorithm proposed in [2]. We approach here the convergence of the algorithm for the case when the constant b from this algorithm is a certain real number.

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

By this modification the fuzzy relaxation algorithm becomes capable of separating two fuzzy linearly non-separable classes.

2. The Fuzzy Relaxation Algorithm

Let $X = (x^1,...,x^p)$, $x^j \in \mathbf{R}^s$ a data set and $\{A_1,A_2\}$ a fuzzy binary partition on X. We will consider the vectors y^j, obtained from x^j by adding a (s+1)-th component equal to 1.

We consider the sign normalization [2]

$$z = \begin{cases} y & \text{if } A_1(y) \geq 0.5, \\ -y & \text{if } A_2(y) > 0.5. \end{cases}$$

We will denote by Z the set of normalized vectors and will consider A_1 and A_2 as fuzzy sets on Z.

The Fuzzy Relaxation Algoritm [1,2] produces (in certain given conditions) a unitary separation vector v satisfying

$$v^T z \geq b > 0, \tag{1}$$

where b is a real positive number.

The correction rule used by the algorithm is (see [2])

$$v^{k+1} = \begin{cases} v^k + c(A_i(z^k))^2 \dfrac{b - v^{kT} z^k}{\|z^k\|^2} z^k & \text{if } A_i(z^k) > 0.5 \text{ and } v^{kT} z^k \leq b, \\ v^k & \text{otherwise.} \end{cases} \tag{2}$$

Instead of using the separation condition (1), as it appears in [2], we will use here a slightly different separation condition, namely:

$$v^T z \geq b, \; b \in \mathbf{R}. \tag{1'}$$

As we will see in the next section, the separation condition (1') allows the algorithm to work in the non-separable case.

3. Study of the properties of this algorithm

Firstly, let us introduce some important notations:

$$E(v,A) = \min \{ v^T z \mid z \in Z \}$$
$$E(A) = \sup \{ E(v, A) \mid \|v\| = 1, v \in \mathbb{R}^{n+1} \} \quad (3)$$
$$V^*(b) = \{ v \mid \|v\| = 1, v^T z > b \ \forall z \in Z \}$$

The following theorem shows a link between E(A) and the linear separability of the fuzzy sets A and $\bar{A}$:

Theorem 1. Let X be a data set and A a fuzzy set on X. The following statements are equivalent:

(i) $E(A) > 0$;

(ii) A and $\bar{A}$ are linearly separable fuzzy sets.

Proof. For the first part of the proof, let us see that $E(A)>0$ implies that there exists a certain v so that $E(v, A) > 0$. Let us denote the quantity $E(v, A)$ by b. From the definitions (3) we may deduce that $v^T z \geq b > 0$ for all $z \in Z$, and from here, the linear separability of A and $\bar{A}$.

Conversely, if A and $\bar{A}$ are linear separable, then there exists a vector v so that $v^T z > 0$ for all $z \in Z$. Let us denote

$$v' = v / \|v\|.$$

Thus, we have that $E(v', A) > 0$, and from here it occurs that $E(A) > 0$. □

The following theorem shows a condition for the existence of the set $V^*(b)$ of solution vectors:

Theorem 2. Let X be a data set, A a fuzzy set on X and b a real number. The following statements are equivalent:

(i) $b \geq E(A)$;

(ii) $V^*(b) = \varnothing$.

Proof. For the first part of the proof let us suppose that $b \geq E(A)$. It implies that for all the unitary vectors v, $E(v,A) \leq b$. Thus, for each unitary vector v, there exists at least a $z \in Z$ such that $v^T z \leq b$, and from here we deduce that $V^*(b) = \emptyset$.

Conversely, let us consider the unitary vector v as fixed. Thus, there exists at least a $z \in Z$ such that $v^T z \leq b$, and from here we have that $E(v, A) \leq b$. As the vector v was previously considered fixed, this propriety is valid for all the unitary vectors v. Thus, we conclude that $E(A) \leq b$. □

The following proposition gives a few properties of the se $V^*(b)$:

Proposition. Let X be a data set, A a fuzzy set on X and b a real number. The following statements are valid (Int V denotes the interior part of the set V, and Fr V denotes the border of the set V):

(i) $\text{Int } V^*(b) = \{v \mid \|v\| = 1, \forall z, v^T z > b\} = V^*(b)$;

(ii) $\text{Fr } V^*(b) = \{v \mid \|v\| = 1, \forall z, v^T z \geq b \text{ and } \exists z: v^T z = b\}$.

Proof. (i) Let us consider the family of sets

$$M_z = \{v \mid \|v\| = 1, v^T z > b\}, z \in Z.$$

Thus it is clear that

$$V^*(b) = \cap \{M_z \mid z \in Z\}.$$

But, M_z is an open set, and thus, $V^*(b)$, being a finite intersection of open sets, is an open set.

(ii) Let us denote

$$M = \{v \mid \|v\| = 1, \forall z, v^T z \geq b \text{ and } \exists z : v^T z = b\}.$$

Let us consider a v^* in M. So, we may split the set Z into Z_1 and Z_2 so that

$$(v^*)^T z > b, \forall z \in Z_1$$

and

$$(v^*)^T z = b, \forall z \in Z_2.$$

Let us chose an ε so that the sphere $S(v^*, \varepsilon)$ verifies the property

$$\forall\, v \in S(v^*, \varepsilon),\ \forall\, z \in Z_1,\ v^T z > b. \tag{4}$$

Due to the first part of this proposition, such an ε does certainly exist.

The hyperplanes $(v^*)^T z = b$ for $z \in Z_2$ split the sphere into a finite number of distinct regions. Important for us are only two of these regions, let us denote them R_1 and R_2, that verify

$$\forall\, v \in R_1,\ \forall\, z \in Z_2,\ v^T z > b \tag{5}$$

and

$$\forall\, v \in R_2,\ \forall\, z \in Z_2,\ v^T z < b. \tag{6}$$

From the relations (4), (5) and (6) it is clear that every v in R_1 is inside $V^*(b)$ and every v in R_2 is outside $V^*(b)$, and that proves that $M = \mathrm{Fr}\, V^*(b)$. This concludes the proof. □

From this proposition we may deduce the following

Corolary. Let X be a data set, A a fuzzy set on X and b a real number. The following equivalences are valid:

(i) $v \in \mathrm{Int}\, V^*(b) \Leftrightarrow E(v, A) > b$;

(ii) $v \in \mathrm{Fr}\, V^*(b) \Leftrightarrow E(v, A) = b$.

Proof. Is very easy as is based on the definition of E(v, A) and the Proposition above. □

Let us denote by R(b) the separation vector produced by the Fuzzy Relaxation Algorithm (as modified in this paper) under the condition $v^T z > b$, when this separation vector does exist.

The following theorem presents a convergence condition of the sequence (v^n) prodced by the Fuzzy Relaxation Algorithm. It represents a generalization of the convergence theorem given in [4]:

Theorem 3. Let X be a data set, A a fuzzy set on X and b and c two real numbers. Let (v^n) be the sequence of the vectors produced by the Fuzzy Relaxation Algorithm under the condition $v^T z > b$. If $0<c<2$ and $b<E(A)$, then the sequence (v^n) is convergent.

The proof of the convergence theorem as it has been stated in [3,4] is applicable even for the supplementary conditions imposed to b. Moreover, the proof of the theorem from [3,4] is based, even if not clearly specified, on a condition similar with $b < E(A)$.

Theorem 4. On the notation from the Theorem 3, if $b < E(A)$ and the sequence (v^n) is finite, then $E(R(b), A) > b$.

Proof. Since (v^n) is finite, there exists a certain $N \geq 1$ such that for all the i's with $i \geq N$, $v^i=v^N$. So, $v^N=v^{N+1}=...=R(b)$, and that implies $(R(b))^T z^i > b$ for every i, and thus $R(b) \in V^*(b)$. Finally, we have that $E(R(b), A) > b$, and that concludes the proof. □

Theorem 5. On the notations from the Theorem 3, if $b < E(A)$, $0<c<2$ and the sequence (v^n) is infinite, then $E(R(b), A) = b$.

Proof. Let us remember that

$$R(b) = \lim_{n \to \infty} v^n.$$

We must show the following:

(i) for every i, $R(b)^T z^i \geq b$;

(ii) there exists at least an i such that $R(b)^T z^i = b$.

For the first part, let us consider the correction rule (2). Since R(b) is the limit of the sequence (v^n), it results that the correction rule do not modifies its value. So, we have the cases:

(a) There exists a certain i such that $R(b)^T z^i \geq b$. It results that

$$R(b) = R(b) + c((A_j(z^i))^2 \frac{b - R(b)^T z^i}{\|z^i\|^2} z^i,$$

where $A_j(z^i) > 0.5$, and, from here,

$$c\,(b - R(b)^T z^i) = 0$$

and, finally,

$$R(b)^T z^i = b.$$

(b) For the rest of the i's, we have that $R(b)^T z^i > b$, and the correction rule lets R(b) unmodified.

So, for all the i's $R(b)^T z^i \geq b$.

For the second part, if we had $R(b)^T z^i > b$ for all the i's, we would have that $R(b)^T z^i \in$ Int $V^*(b) = V^*(b)$. Since every vicinity of R(b) contains at least an element v^j of the sequence (v^n), it results that there exists a $v^j \in V^*(b)$. Thus, v^j is a stop point and (v^n) is a finite sequence, and that contradicts the hypothesis.

Finally, we have that $R(b) \in$ Fr $V^*(b)$ and that $E(R(b), A) = b$. That concludes the proof. □

3. Concluding remarks

It is certainly interesting to study what happens in the case $b \geq E(A)$. Even if we haven't proved yet, the experience enables us to consider the following

Conjecture. On the notations from the Theorem 3, if $b \geq E(A)$ the sequence (v^n) is convergent and $E(R(b), A) = E(A)$.

Let us notice that we did not introduce any restriction with respect to $b > 0$. Consequently, the theorems presented above are valid for the case $b < 0$, with the single condition that $b < E(A)$. So, we may assure the output of a 'separation' hyperplane for the case of linear non-separability, i.e. when $b < E(A) < 0$. This is a remarkable property of the Fuzzy

Relaxation Algorithm.

Other interesting problem is whether there exists a modality to compute directly E(A) and whether there exists a certain v such that E(v,A) = E(A). Thus, the Fuzzy Relaxation Algorithm would be able to produce the optimal separation hyperplane with respect to E(A).

REFERENCES

1. D. Dumitrescu. A fuzzy training algorithm. Studia Universitatis "Babeş-Bolyai", Ser. Math. 35, (1990), 7-12.

2. D. Dumitrescu. Fuzzy training procedures, I. Fuzzy Sets and Systems 56 (1993), 155--169.

3. D. Dumitrescu. Principiile matematice ale teoriei clasificării, Editura Academiei Române, Bucureşti, 1995.

4. D. Dumitrescu. Fuzzy sets and their applications for clustering and training, to appear.

5. D. Dumitrescu, Horia F. Pop. Convex decomposition of fuzzy partitions, I,II. Fuzzy Sets and Systems (1995). to appear.

6. J. Sklansky, and G. N. Wassel. Pattern Classifiers and Trainable Machines. Springer Verlag, New York, 1981.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

LOGICAL GRAMMARS AND UNFOLD TRANSFORMATION OF LOGIC PROGRAMS

Doina TATAR*

Dedicated to Professor Iever Groze on his 65th anniversary

Received: June 28, 1994
AMS subject classification: 68Q50, 68Q55, 68T27

REZUMAT. - Gramatici logice şi transformări "unfold" ale programelor logice. În articol este utilizată noţiunea anterior definită în [12] de "gramatică logică" pentru a demonstra faptul că transformările "unfold" ale programelor logice conduc la programe care sunt echivalente cu cele originale.

Abstract. The previously defined notion of "logical grammar" [12] is utilized to demonstrate that unfold transformations of logic programs produce programs which are equivalent to the original one.

1. Introduction. The two principal papers which we are based in this note are:" Logical grammars as a tool for studying logic programming" [12] (where we propose a new semantic for operational behavior of logic programs, which is based on the framework of formal languages), and "Unfold /fold transformations of logic programs" by J.C.Shepherdson [10]. In a way, this paper is the first certificate (besides those from [12]) that this new approach for semantic of logic programs is better matching for some connected discussions.

We start by looking at the unfold transformations : the origin of those goes back to Burstall and Darlington,who introduced them in the context of recursive programs. In [10] the definition of unfolding for a logic program is:

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

Definition 1.1 Let P be a logic program and

$$C : A \leftarrow p(t), S$$

be a clause of program P (where t stands for a tuple of terms) and

$$D_1 : p(t_1) \leftarrow S_1$$

$$D_r : p(t_r) \leftarrow S_r$$

be all the clauses of P whose heads $p(t_1),...,p(t_r)$ unify with p(t) with mgu $\theta_1,...,\theta_r$. Then the result of unfolding C w.r.t. p(t) is the program P' obtained by replacing C by the r clauses

$$C_1 : A\theta_1 \leftarrow S_1\theta_1, S\theta_1$$

$$C_r : A\theta_r \leftarrow S_r\theta_r, S\theta_r$$

Here A is an atom, p is a symbol of predicate and S is the rest of a clause.

In [8] is obtained the result:

Theorem If P' is the logic program obtained from P by unfolding, then a goal G success with the answer substitution θ from P' iff the goal G success with the answer substitution θ from P.

In the next section we will present the logical grammars [12] and will prove that the unfold transformation preserve the meaning introduced by those.

2. Logical grammars The language considered here is essentially that of first-order predicate logic without function symbols. Let:

- P be a set of predicates.
- C be a set of constants.
- V be a set of variables.

An atom over $C \cup V$ is of the form

$$p(u_1,...,u_n),\ n \geq 0$$

where $p \in P$ with arity n, and each u_j is an element of $C \cup V$.

If the arguments u_j are not interesting in a particular context, then we will denote an atom simply by p. Let A be the set of atoms over $C \cup V$, and, if

$P' \subseteq P$, let $A_{P'}$ be the set of atoms with predicate symbols from P'. In some recent papers [13], the set of predicates is considered as divided into two disjoint sets:the set EDB of extensional predicates (or extensional databases predicate) which represent basic "facts," and the set IDB(of intensional databases predicates) representing facts deduced from the basic facts via the logic program. Particularly ,the set EDB can be the empty set (as,for simplicity,in most of the following demonstrations).

Definition 2.1 A logic program P is a sequence of Horn clauses, that is, clauses of the form:

$$p \leftarrow q_1,\ldots,q_n$$

where p and $q_1,\ldots,q_n$ are atomic formulas in first-order logic, the comma is the logic operation "and", and the sign $\leftarrow$ is "if" or reverse of the logical implication.

We refer to the left (p) and right-hand side ($q_1,\ldots,q_n$) of a clause as its head and body. A clause is logically interpreted as the universal closure of the implication $q_1 \wedge \ldots \wedge q_n \rightarrow p$. If a clause has no right-hand side we will call it a fact or a unit clause. Let us observe that this definition considers only the class of positive logic programs (all atoms in all clauses are positive).
From the properties of IDB and EDB predicates it follows that a predicate from the set EDB cannot occur in the head of clauses, but a predicate from the set IDB can occur in the set of facts.

Definition 2.2 A goal G consists of a conjunction of atoms, such that a successfully terminating computation corresponds to a demonstration of this goal by refutation (SLD-refutation), and is denoted by:

$$\leftarrow r_1,\ldots,r_t$$

Over the course of time, we may want to "apply" the same IDB for many quite different EDB's. In this context, and because IDB is the "core" of logic program, the properties of the IDB merit careful study. Recent papers have addressed the problems of studying and optimizing logic programs from various points of view. ([2], [3], [4]). In [12] our tool is

the logical grammars:

Definition 2.3 The logical grammar GL associated with a logic program P and the goal G is the system:

$$GL = (I_N, I_T, X_0, F)$$

where:

- $I_N = A_{IDB} \cup \{X_0\}$ is the set of nonterminals.
- $I_T = A_{EDB} \cup \{\lambda\} \cup \{False, True\}$ is the set of terminals.
- X_0 = is the goal G.
- F is a finite set of production rule, of the form:

$$a)\ p \rightarrow q_1 \ldots q_m,\ m \geq 1$$

where $p \in IDB$ and $p \leftarrow q_1,\ldots,q_m$ is a clause in the program P.

or

$$b)\ p \rightarrow \lambda$$

where p is a unit clause in the program P

We assume in the following that substitutions, composition of substitutions, and the most general unifier $\sigma = mgu(g,h)$ of atoms g and h are defined as in logic programming. ([1], [2], [3]).

For a logic grammar GL we define the rewriting relation " $\Rightarrow$ " as follows:

Definition 2.4 If $R \in A^+$ and $Q \in A^*$, then:

$$R \Rightarrow_{GL}^{\sigma} Q$$

if exist an atom $h \in I_N$, and a production rule in F:

$$g \rightarrow h_1 \ldots h_m$$

such that:

$$R = R_1 h R_2, \sigma = mgu(h,g)$$

and

$$Q = \sigma(R_1)\sigma(h_1)\ldots\sigma(h_m)\sigma(R_2)$$

(here the variables of the production rule are renamed to new variables, so that all the variables in the rule do not appear in R).

Let $\Rightarrow^*$ denote the reflexive and transitive closure of the relation $\Rightarrow$. If θ is the composition of all substitutions in every direct derivation, let denote it by $\Rightarrow^{\theta}$ *.

Definition 2.5 For a logical grammar $GL = (I_N, I_T, X_0, F)$, the generated language is L(GL), where:

$L(GL) = \{(R,\theta) | X_0 \Rightarrow^{\theta} {}^*R, R \in A^*_{IT}, \theta = \theta_1 \ldots \theta_k$, k is the length of derivation for R, and θ_i is the substitution in the step i$\} \cup \{\Omega\}$.

We have some possibilities for the pair (R, θ):

- if X_0 (or the goal G) is a ground formula, (not containing the variables) then the substitution θ is the empty substitution, and R is True or False, depending on the fact that G is a formula deducible or not from the set of clauses P (by refutation).

- if X_0 contains variables, and the computation is succesfully terminating, in the pairs (R,θ) we have $R \in I^*_T$, and the number of pairs represents the number of solutions. If EDB = ϕ ,then R = λ. Let denote the last situation
by R= [] ,the empty clause, like usually in logic, and let θ be the answer substitution.

- if the program P is not terminating for the goal G, then $L(GL) = \{\Omega\}$, where $\Omega \notin$ IDB $\cup$ EDB.

For the purpose of simpler manipulation, let us denote the logical grammar GL with the symbol initial X_0 by GL_{X0} and the pairs in the logical language
$L(GL_{X0})$ by the triplets (P,θ,X_0). Let us, furthermore suppose EDB = ϕ. Then the deflnition (2.5) becomes:

$$L(GL_{X0}) = \{([],\theta,X_0) | X_0 \Rightarrow^{\theta} {}^* []\} \cup \{\Omega\}$$

In [5] the "succes set " for a logic program P is defined as:

succ(P) = {$G(t_1,\ldots,t_n | t_1,\ldots,t_n$ are terms in a standard Herbrand interpretation, then they are grounded terms, and the goal $G(t_1,/cdots,t_n)$ is deductible from P} or, equivalentely,

succ(P) = {$G(t_1,\ldots,t_n | t_1,\ldots,t_n$ are terms in a standard Herbrand interpretation and there exists a SLD-refutation of $G(t_1,\ldots,t_n)$ from P}

The set succ(P) is not completely adequate as operational semantics since it hides one of the fundamental aspects of logic programming: the ability to compute substitutions. A more adequate definition is [12] the following:

succ'(P) = $\{(G(t_1,...,t_n),\theta) | t_1,...,t_n$ are non-ground terms, and $G(t_1,...,t_n)$ has a SLD-refutation with computed answer $\theta\}$.

In [12] we proved the connection between the set succ'(P) and the logical grammars.

Definition 2.6 Let U'_H be a nonstandard Herbrand interpretation (which admits non-ground terms). We denote by L_p and we will call them the total language of a logical program P, the following language:

$$L_p = \bigcup_{X_0 \in U'_H} L(GL_{X_0})$$

Conformally with the previously definitions,

$L_p = \{([],\theta,G) | G$ is an arbitrary goal, $G \Rightarrow^{\theta *} []\} \cup \{(\Omega,G) |$ P is cycling for the goal G$\}$ $\cup$ {(False,G)|G has not a SLD-refutation from P}. In [12] we demonstrated that this language L_p represents a sound and complete semantic for P:

Lema If $G \Rightarrow^{\sigma} G'$ and $(G',\theta) \in$ succ'(P) then $(G,\sigma\theta) \in$ succ'(P).

Theorem (of soundness) Let P be a logic program and G a goal. If $([],\theta,G) \in L_p$ then $(G,\theta) \in$ succ'(P).

Theorem (of completeness) Let P be a logic program and G a goal. If $(G,\theta) \in$ succ'(P), then $([],\theta,G) \in L_p$.

For a logic program P let denote by GL_p the associated logic grammar like in definition (2.5).

We can define two kinds of equivalence of the logic programs:

Definition Two logic programs P_1 and P_2 are strong equivalent if $\forall X_0$ we have $L(GL_{P1}) = L(GL_{P2})$ (or, in the formal languages terminology, if GL_{P1} and GL_{P2} are equivalent for every goal X_0).

The equivalence of two programs in the following is the strong equivalence, therefore the equivalence for same goal.

Definition. Two logic programs P_1 and P_2 are equivalent if $L_{P1} = L_{P2}$ where L_P is defined like in definition 2.6.

The consequence of the introduced notion is the possibility of the definition for some transformations about logic programs, such that the obtained programs are equivalent with the initial programs.

The main result of this section is the following theorem;

Theorem If P' is a logic program obtained from P by unfolding, then P and P' are strong equivalent.

Proof It is enough to prove that, if:

$$Q,R \in A^*_{I_T \cup I_N},\ Q \Rightarrow_{GL_P} {}^*R$$

then

$$Q \Rightarrow_{GL_{P'}} {}^* R$$

where GL_p and $GL_{p'}$ are the logic grammars associated with the programs P and P'. Let observe that ,in accordance with the definition 1.1 and 2.3, in GL_p there exist the production rules:

$$1.\ A \rightarrow p(t)S$$

$$2.\ p(t_1) \rightarrow S_1$$

$$p(t_r) \rightarrow S_r$$

and in $GL_{p'}$ these became

$$3.\ A\theta_1 \rightarrow S_1\theta_1 S\theta_1$$

$$A\theta_r \rightarrow S_r\theta_r S\theta_r$$

Also, conformaly with definition 2.4, if $Q \Rightarrow_{GL_P} {}^* R$ where a rule 1 followed by 2 is utilized, then $Q \Rightarrow_{GL_{P'}} R$ by a single rule 3. The grammar GL_p has the same generative power like $GL_{p'}$, thus P and P' are strong equivalent.

REFERENCES

1. K.R. Apt, M.H.van Emden: "Contribution to the theory of logic programming", J. of ACM, vol.29, 1982, pg.841-862.
2. K.R. Apt, D. Pedreschi: Studies in pure Prolog:termination, CWI Report CS-R9048, September, 1990.
3. K.R. Apt, D. Pedreschi: Proving termination of general Prolog programs, CWI Report CS-R9111, February, 1991.
4. S.Debray, P.Mishra:"Denotational and operational semantics for PROLOG", The Journal of Logic Programming, vol.5, nr.1, 1988, pp.33-61.
5. M.Falaschi, G.Levi, M.Martelli, G.Palamidessi:"Declarative modelling of the operational behaviour of logic languages", Raport Univ.di Pisa,TR-10/1980.I.Guessarian: Some fixpoint techniques in algebraic structures
6. P.A.Gardner, J.C.Shepherdson:"Unfold/fold transformation of LP", Festschrift in Honour of Alan Robinson, Oxford University Press,London, 1990.
7. C.J. Hogger: "Derivation of Logic Programs", Journal of ACM, April, 1981, pp. 372-393.
8. T.Przymusinski:"On the declarative and procedural semantics of logic programs" J.of Automated Reasoning, vol5, 1989, pp.167-205.
9. I. Shioya: "Logic hypergraph grammars and context-free hypergraph grammars". Systems and Computers, vol.22, nr.7, 1991.
10. J.C.Shepherdson:"Unfold/fold transformations of LP ", Mathem. Struct.in Computer Science(1992), vol.2, pp.143-157.
11. D.Tatar:" Utilizarea gramaticii sintactice asociata unei scheme de recursive in studiul programelor", Studii şi cercetari matematice, nr.4, 1988, pp.337-347.
12. D.Tatar:"Logical grammars as a tool for studying logic programming", Studia Univ."Babes-Bolyai", Mathematica, 1993(to appear).
13. A. van Gelder, K.A.Ross, J.S.Schlipf:"The Well-Founded Semantics for General Logic Programs", Journal of ACM, July, 1991, pp. 620-651.
14. M.H.van Emden, R.A.Kowalski:"The semantics of predicate logic", J.of ACM, oct.1976, pp.733-742.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

LOGICAL GRAMMARS AS A TOOL FOR STUDYING LOGIC PROGRAMMING

Doina TATAR*

Dedicated to Professor Sever Groze on his 65th anniversary

Received: March 11, 1995
AMS subject classification: 68Q50, 68Q55, 68T27

REZUMAT. - Gramatici logice ca instrument în studiul programării logice. Articolul defineşte o nouă semantică operaţională a programelor logice, semantică bazată pe limbaje formale şi pe interpretări care pot conţine atomi cu argumente variabile. Este demonstrată corectitudinea şi completitudinea acestei semantici. În acest mod, anumite rezultate bine cunoscute în teoria limbajelor formale pot fi utilizate în studiul programelor logice.

Abstract. The paper defines a new operational semantics for logic programs, which is based on framework of formal languages and on interpretations containing possibly non-ground atoms. The soundness and completness are shown to hold. Thus, some folklore results in formal languages can be utilized for improvement of logic programs.

1. Introduction. In recent years there has been a great deal of interest in logic programming languages. The first step in this development was the seminal paper by van Emden and Kowalski [14],in which they outlined declarative and operational semantics of Horn Clause Logic (HCL) as a programming language.Apt and van Emden [1] built upon the former work and defined the fixpoint semantic of HCL.From one point of view,any attempt at formulating a semantic for a program P in a logic programming language (as PROLOG) is simply closed,as programs are statements in the HCL fragment of first-order logic. On the other hand, from a computational point of view, this semantics is not completely adequate, as they ignore several behavioural aspects of logic programs. These

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

include issues such as termination, the answer substitution, the search strategy, etc. This is the reason for which many authors attempt to define new semantics for logic programs: [5], [6], [10], [13].

In this paper we propose a new semantic for operational behaviour of logic programs, which is based on the framework of formal languages. Several direct correspondences that can be made for transforming and optimising logic programs are pointed out. In non-monotonic logic programming this framework is very useful also. We think that an deeper connexion between these two fields, apparently so different, can be to benefit by logic programming:the formal languages(tree-languages) and the rewriting systems are very intensively studied since many years.

2. Preliminaries.

The language considered here is essentially that of first-order predicate logic without function symbols. Let:

- P be a set of predicates.
- C be a set of constants.
- V be a set of variables.

An atom over $C \cup V$ is of the form

$$p(u_1,...,u_n),\ n \geq 0$$

where $p \in P$ with arity n, and each u_j is an element of $C \cup V$.

If the arguments u_j are not interesting in a particular context, then we will denote an atom simply by p. Let A be the set of atoms over $C \cup V$, and if $P' \subseteq P$ let $A_{P'}$ be the set of atoms with predicate symbols from P'. In some recent papers [7], [9], [13], the set of predicates is considered as divided into two disjoint sets:the set EDB of extensional predicates (or extensional databases predicates) which represent basic "facts", and the set IDB(of intensional databases predicates) representing facts deduced from the basic facts via the logic program. Particularly, the set EDB can be the empty set (as, for simplicity, in most of the following demonstrations).

Definition 2.1 A logic program P is a sequence of Horn clauses, that is, clauses of the form:

$$p \leftarrow q_1,\ldots,q_n$$

where p and $q_1,\ldots,q_n$ are atomic formulas in first-order logic, the comma is the logic operation "and", and the sign $\leftarrow$ is "if" or reverse of the logical implication.

We refer to the left (p) and right-hand side ($q_1,\ldots,q_n$) of a clause as its head and body. A clause is logically interpreted as the universal closure of the implication $q_1 \wedge \ldots \wedge q_n \rightarrow p$. If a clause has no right-hand side we will call it a fact or a unit clause. Let us observe that this definition considers only the class of positive logic programs(all atoms in all clauses are positive).

From the properties of IDB and EDB predicates it follows that a predicate from the set EDB cannot occur in the head of clauses, but a predicate from the set IDB can occur in the set of facts.

Definition 2.2 A goal G consists of a conjunction of atoms, such that a successfully terminating computation corresponds to a demonstration of this goal by refutation (SLD-refutation), and is denoted by:

$$\leftarrow r_1,\ldots,r_i$$

3. Logical grammars

Over time, we may want to "apply" the same IDB for many quite different EDB's. In this context, and because IDB is the "core" of logic program, the properties of the IDB merit careful study. Recent papers have addressed the problems of studying and optimizing logic programs from various points of view. ([2], [3], [4], [7]) Our tool is the logical grammars:

Definition 3.1 The logical grammar GL associated with a logic program P and the goal G is the system:

$$GL = (I_N, I_T, X_0, F)$$

where:

- $I_N = A_{IDB} \cup \{X_0\}$ is the set of nonterminals.
- $I_T = A_{EDB} \cup \{\lambda\} \cup \{False, True\}$ is the set of terminals.
- X_0 is the goal G,
- F is a finite set of production rule, of the form:

a) $p \rightarrow q_1 \ldots q_m$, $m \geq 1$

where $p \in IDB$ and $p \leftarrow q_1,\ldots,q_m$ is a clause in the program P.

or

b) $p \rightarrow \lambda$

where p is a unit clause in the program P.

We assume in the following that substitutions, composition of substitutions, and the most general unifier σ = mgu (g,h) of atoms g and h are defined as in logic programming. ([1], [2], [3], [12]).

For a logic grammar GL we define the rewriting relation " $\Rightarrow$ " as follows:

Definition (3.2) If $R \in A^+$ and $Q \in A^*$, then:

$$R \Rightarrow^{\sigma}_{GL} Q$$

if exists an atom $h \in I_N$, and a production rule in F:

$$g \rightarrow h_1 \ldots h_m$$

such that:

$$R = R_1 h R_2, \sigma = mgu\ (h,g)$$

and

$$Q = \sigma(R_1)\ \sigma(h_1)\ldots\sigma(h_m)\sigma(R_2)$$

(here the variables of the production rule are renamed to new variables ,so that all the variables in the rule do not appear in R).

Let $\Rightarrow^*$ denote the reflexive and transitive closure of the relation $\Rightarrow$. If θ is the composition of all substitutions in every direct derivation,let denote it by $\Rightarrow^{\theta}$ *.

Definition (3.3) For a logical grammar GL = (I_N, I_T, X_0, F) the generated language is L(GL), where:

$L(GL) = \{(R,\theta) \mid X_0 \Rightarrow^{\theta} * R, R \in A^*_{I_r}$, $\theta = \theta_1 ... \theta_k$ is the length of derivation for R, and θ_i is the substitution in the step i$\} \cup \{\Omega\}$.

We have some possibilities for the pair (R,θ):

- if X_0 (or the goal G) is a ground formula, (not containing the variables) then the substitution θ is the empty substitution, and R is TRUE or FALSE, depending on the fact that G is a formula deducible or not from the set of clauses P (by refutation).
- if X_0 contains variables, and the computation is succesfully terminating, in the pairs (R,θ) we have $R \in I^*_T$, and the number of pairs represents the number of solutions. If EDB= ϕ, then $R = \lambda$. Let denote the last situation by R=[], the empty clause, like usually in logic,and let θ the answer substitution.
- if the program P is not terminating for the goal G, then $L(GL) = \{\Omega\}$, where $\Omega \notin$ IDB $\cup$ EAB.

Example

Let P be the program:

```
domains
        lista = symbol*
predicates
        m(symbol, integer, lista)
        consec(symbol, symbol, lista)
clauses
        consec(U,V,X):-m(U,I,X),m(V,I+1,X).
        m(U,1,U.X).
        m(U,I+1, V.X):-m(U,I,X).
```

If the goal is: G=consec (c,X,a.b.c.d.nil)

then the derivation is:

consec(c,X,a.b.c.d.nil) $\Rightarrow^{\theta 1=\lambda}$ m(c,I,a.b.c.d.nil)m(X,I+1,a.b.c.d.nil)$\Rightarrow^{\theta 2=I/I'+1)}$ m(c,I', b.c.d.nil)
m(X,I'+2,a.b.c.d.nil)$\Rightarrow^{\theta 3=(I'/I''+1)}$ m(C,I",c.d.nil)m(X,I"+3,a.b.c.d.nil)$\Rightarrow^{\theta 4=(I''/1)}$
m(X,4,a.b.c.d.nil)$\Rightarrow^{\theta 5=\lambda}$ m(X,3,b.c. d.nil)$\Rightarrow^{\theta 6=\lambda}$ m(X,2,c.d.nil) $\Rightarrow^{\theta 7=\lambda}$ m(X,1,d.nil)$\Rightarrow^{\theta 8=(X/d)}\lambda$.

The substitution in variable X only is:

$$\theta = \theta_1\theta_2...\theta_g = (X/d)$$

and the pair $([],\theta) \in L(GL)$.

4. Remarks about soudness and completness of logical grammars

For the purpose of simpler manipulation , let us denote the logical grammar GL with the symbol initial X_0 by GL_{X_0} and the pairs in the logical language $L(GL_{X_0})$ by the triplets (P,θ, X_0). Let us, furthermore suppose EDB=ϕ. Then, the definition (3.3) becomes :

$$L(GL_{X_0}) = \{([],\theta,X_0)|X_0 \Rightarrow^{\theta *} []\} \cup \{\Omega\}$$

In [6] the "success set " for a logic program p is defined as:

succ(P) = {$G(t_1,...,t_n)|t_1,...,t_n$ are terms in a standard Herbrand interpretation, the.. they are grounded terms, and the goal $G(t_1,...,t_n)$ is deductible from P}

or, equivalently,

succ(P) = {$G(t_1,...,t_n)|t_1,...,t_n$ are terms in a standard Herbrand interpretation and there exists a SLD-refutation of $G(t_1,...,t_n)$ from P}

The set succ(P) is not completely adequate as operational semantics since it hides one fundamental aspect of logic programming: the ability to compute substitutions. A more adequate definition should be the following:

succ'(P) = {$(G(t_1,...,t_n),\theta)|t_1,...,t_n$ are non-ground terms, and $G(t_1,...,t_n)$ has a SLD-refutation with computed answer θ}.

Let us remark that in [6] two other models (S-model and C-model) are introduced by permitting that $t_1,...,t_n$ are not necessarily ground terms.

In the sequel we will see the connexion between the set succ'(P)and the logical grammars introduced in paragraph 3.

Definition 4.1 Let U'_H be a nonstandard Herbrand interpretation (which admits non-ground terms). We denote by L_μ and we will call them the total language of a logical program P, the following language:

$$L_p = \bigcup_{\chi_0 \in U'_H} L(GL_{\chi_0})$$

Conformally with the previous definitions,

L_p={([],θ,G)|G is an arbitrary goal, $G \Rightarrow^{\theta} * []$}∪{(Ω,G)|P is cycling for the goal G} ∪ {(False,G)|G has not a SLD-refutation from P}. Let prove that this language L_p represents a sound and complete semantic for P.

Lema 4.2

If $G \Rightarrow^{\sigma} G'$ and $(G',\theta) \in succ'(P)$ then $(G,\sigma\theta) \in succ'(P)$.

Demonstration

We denote shortly for $(G',\theta) \in succ'(P)$ by $P \rightarrow G'\theta$, where " → " is logical implication and G'θ is the conjunction of atoms of the goal G'θ.

Suppose $G = A_1,...,A_s,...,A_k$ and

$G' = A_1\sigma,...,A_{s-1}\sigma,B_1\sigma,...B_m\sigma, A_{s+1}\sigma,...,A_k$ such that the rule

$A \rightarrow B_1...B_m$ was applied in the rewriting $G \Rightarrow G'$, with $\sigma = mgu(A_s,A)$.

By logical means of applied rule results that $B_1 \wedge ... \wedge B_m \rightarrow A$

and by syllogism we have sequently :

$P \rightarrow G'\theta$, and $G'\theta \rightarrow A_1\sigma\theta \wedge ... \wedge A_{s-1}\sigma\theta \wedge A_s\sigma\theta \wedge ... \wedge A_k\sigma\theta$

Results that $P \rightarrow G\sigma\theta$, such that $(G,\sigma\theta) \in succ'(P)$.

Theorem 4.3 (of soundness)

Let P be a logic program and G a goal. If $([],\theta,G) \in L_p$ then $(G,\theta) \in succ'(P)$.

Demonstration Let $G = A_1,...,A_k$ and assume that $([],\theta,G) \in L_p$

We prove by induction on the length n of deduction of ([],θ,G) that $(G,\theta) \in succ'(P)$. If n=1 then exists a unit clause A_1 in P such that $A_1 \Rightarrow^{\theta 1} []$.

Then $(\theta_1,A_1) \in succ'(P)$.

Let $G \Rightarrow^{\theta} * []$ be a deduction of length n. Then it can be detailed as:

$$G \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta_1} ... \Rightarrow^{\theta_n} []$$

or

$$G \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta'} * []$$

where $G_1 \Rightarrow^{\theta'} * []$, $\theta' = \theta_2 \ldots \theta_n$ is a deduction of length n-1.

Suppose that $G = A_1,\ldots,A_s,\ldots,A_k$

and $G_1 = A_1\sigma,\ldots,A_{s-1}\sigma, B_1\sigma,\ldots,B_m\sigma,A_{s+1}\sigma,\ldots,A_k$

where $\sigma = mgu(A_s,A)$, $A \leftarrow B_1,\ldots,B_m$ is a clause in P. (That means

$G \Rightarrow^{\sigma} G_1$, $\theta_1 = \sigma$).

By induction hypothesis and, on the other hand, from $G \Rightarrow^{\theta 1} G_1$ and lema (4.2) we obtain $(G,\theta_1,\theta') = (G,\theta) \in succ'(P)$.

Theorem 4.4 (of completeness)

Let P be a logic program and G a goal. If $(G,\theta) \in succ'(P)$, then

$([],\theta,G) \in L_p$.

Demonstration By induction on the length of a logical implication of $G\theta$ from P.

5. Applications

For a logic program P let denote by GL_p the associated logic grammar like in definition (3.3).

We can define two kinds of equivalence of the logic programs:

Definition (5.1) Two logic programs P_1 and P_2 are strong equivalent if $\forall X_0$ we have $L(GL_{P_1}) = L(GL_{P_2})$ (or, in the formal languages terminology, if GL_{P_1} and GL_{P_2} are equivalent for every goal X_0).

The equivalence of two programs in the following is the strong equivalence, therefore the equivalence for same goals.

Definition 5.2 Two logic programs P_1 and P_2 are equivalent if $L_{P_1} = L_{P_2}$

where L_p is defined like in definition 4.1.

The consequence of the introduced notion is the possibility of the definition for some transformations about logic programs, such that the obtained programs are equivalent with the initial programs.

Definition 5.3 [11]. A logic program P is said to be "constant-free", if the atom in left side of each clause contains no constants.

Theorem 5.4 Every logic program P can be transformed into an equivalent "constant-free" program P', if the below procedure C is not failing.

Proof: Let observe that a program P is "constant-free" if associated logic grammar GL_p is such that in every production rule, the atom in left side (a nonterminal atom) contains no constants. We will apply the following procedure C, while remains a production rule with an atom containing constants in his left side.

Procedure C: Let $a = x_j$ be a constant contained in the nonterminal atom $p(x_1,...x_n)$ in one of his occurs in left side of a production rule.

Rule C1. Replay every production rule of the form:

$$p(y_1,...,y_n) \rightarrow h_1...h_k$$

such that $y_j = a$ by a new production rule

$$p_{j,a}(y_1,...,y_{j-1}, y_{j+1},...,y_n) \rightarrow \sigma(h_1)...\sigma(h_k)$$

where $\sigma = [(y_j/a)]$. or $\sigma = mgu(y_j,a)$

Let observe that we can have, in another rule with nonterminal p in left side, the case $y_j = b$ and b is not a. Rule C1 introduces in this case the new nonterminal $p_{j,b}$ $(y_1,...,y_{j-1},y_{j+1},...y_k$. Then, the rule C1 cannot introduce a failing situation for the procedure C.

Rule C2. Replay every production rule of the form:

$$q \rightarrow l_1...p(y_1,...,y_n)...l_m$$

by a new production rule:

$$\sigma(q) \rightarrow \sigma(l_1)...\sigma(p_{j,a}(y_1,...,y_{j-1},y_{j+1},...,y_n)...\sigma(l_m)$$

where $\sigma = mgu(a,y_j)$, if σ does exists. Unlike rule C1, if yj=b and b is not a, then σ does not exist and procedure C fails. Theorem 5.4 results by induction on the length of a successful derivation. It is enough to prove that, if:

$$Q,R \in A^*_{I \cup I_N}, Q \Rightarrow^* GL_p R$$

then

$$Q' \Rightarrow GL_{p'}{}^* R'$$

where P' is the logic program obtained by successfully application of procedure C ,and Q',R' are obtained from Q and R by replacing a non constant-free predicates p by their corresponding predicate ($p_{j,a}$).

If the length is 1,then $Q' \Rightarrow^* GL_p.R'$ is true, by once application of rule C_1 or C_2.

Suppose that, for every derivation of length n, we have implication verified. Let $Q \Rightarrow^*_{GL_p} R$ be a derivation of length n+1, and let point out the last step :

$$Q \Rightarrow^*_{GL_p} T \Rightarrow R$$

By induction hypothesis, we have $Q' \Rightarrow^*_{GL_{p'}} T'$. If $T \Rightarrow_{GL_p} R$ by utilising a rule \$ C_1 containing the non constant -free predicate p in left hand-side ,then $T = T_1pT_1$ and $R = T_1h_1...h_nT_2$. By induction hypothesis $T' = T'_1p_{j,a}T'_2$, where T'_1 and T'_2 are corresponding to T_1 and T_2. Then:

$$T'_1p_{j,a}T'_2 \Rightarrow T'_1\sigma(h_1)...\sigma(h_k)T'_2$$

and $R' = T'_1\sigma(h_1)...\sigma(h_k)T'_2$ is corresponding to R .

If $T \Rightarrow_{GL_p} R$ by utilizing a rule of the form C_2, then $T = .{}_1qT_2$, $R=T_1l_1...p(y_1,...y_n)...l_mT_2$. By induction hypothesis $T' = T'_1\sigma(q)T'_2$ and thus $R' = T'_1\sigma(l_1)...\sigma(p_{j,a})(...))...\sigma(l_m)T'_2$ is corresponding to R.q.e.d.

Another syntactical improvement of a logical program is the elimination of the predicates that have the equal arguments.

Definition 5.5 [11] An atom is "loop-free" if all the arguments are different. A production rule is "loop-free" if the nonterminal atom in the left hand side of this production rule is "loop-free". A logic grammar is "loop-free" if all his production rules are "loop-free". A logic program P is "loop-free" if GL_p is "loop-free". Fortunately, a similar with (5.4) theorem can be proved by induction:

Theorem 5.6 Every logic program P can be transformed into an equivalent "loop-free" program P'.

Also, in a logic grammar, as well as in context-free grammars, we can remove useless predicates, without affecting the generated language.

Definition 5.7 [12] A nonterminal atom p is productive if exists a derivation

$$p \Rightarrow {}^{*}Q,\ Q \in A^{*}{}_{I_T}$$

A nonterminal atom p is deductible if exists a derivation:

$$X_0 \Rightarrow^{*} P,\ P \in A^{*}{}_{I_N \cup I_T}$$

such that P contains p.

A logic grammar GL is reduced if every nonterminat atom is deductible and productive. A logic program P is reduced if the associated logic grammar GL is reduced.

Theorem 5.8 Every logic program P can be transformed into an equivalent reduced program P'.

Proof: By application of the algorithms for obtaining a reduced context-free grammar, to the logic grammar GL_P.

REFERENCES

1. K.R. Apt,M.H.van Emden : "Contribution to the theory of logic programming " J. of ACM, vol.29,1982, pg.841-862.
2. K.R. Apt, D. Pedreschi: Studies in pure Prolog:termination, CWI Report CS-R9048, September, 1990.
3. K.R. Apt, D. Pedreschi: Proving termination of general Prolog programs, CWI Report CS-R9111, February, 1991.
4. S.Debray, D.Waren: "Functional Computation in Logic Programs", ACM Transaction, vol.11, 1989, pg.451-481.
5. S.Debray,P.Mishra:"Denotational and operational semantics for PROLOG", The Journal of Logic Programming, vol.5, nr.1, 1988, pp.33-61.
6. M.Falaschi, G.Levi, M.Martelli, G.Palamidessi: "Declarative modelling of the operational behaviour of logic languages" report Univ.di Pisa,TR-10/1980.I. Guessarian: Some fixpoint techniques in algebraic structures
7. H. Gaifman, H.Mairson,Y.Sagiv,M.Y.Vardi:" Undecidable Optimization Problems for Database Logic Programs", Journal of ACM, July, 1993, pp. 683-714.
8. C.J. Hogger: "Derivation of Logic Programs", Journal of ACM, April, 1981, pp. 372-393.

9. J.Minker:"Perspective in deductive databases" J.of Logic Programming, vol.5, 1988, pp.33-61.
10. T.Przymusinski:"On the declarative and procedural semantics of logic programs" J. of Automated Reasoning, vol5, 1989, pp.167-205.
11. I. Shioya: "Logic hypergraph grammars and context-free hypergraph grammars", Systems and Computers, vol.22, n.:7, 1991.
12. D.Tatar:" Utilizarea gramaticii sintactice asociata unei scheme de recursie, în studiul programelor", Studii si cercetari matematice, nr.4, 1988, pp. 337-347.
13. A. van Gelder, K.A.Ross, J.S.Schlipf.:"The Well-Founded Semantics for General Logic Programs", Journal of ACM, July, 1991, pp. 620-651.
14. M.H.van Emden, R.A.Kowalski;"The semantics of predicate logic", J. of ACM, oct. 1976, pp. 733-742.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

MODELLING DISTRIBUTED EXECUTION IN THE PRESENCE OF FAILURES

Alexandru VANCEA*

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 10, 1995
AMS subject classification: 68Q10, 65Y05

REZUMAT. - Modelarea execuţiei distribuite în prezenţa căderilor. Pe măsura răspândirii tot mai accentuate a sistemelor distribuite şi a utilizării lor de către nespecialişti, atributul toleranţei la erori (*fault tolerance*) în prezenţa unor căderi ale unor servere devine o cerinţă absolut esenţială pentru un sistem de calcul distribuit viabil. Lucrarea prezintă o clasificare şi o modelare matematică a celor mai răspândite tipuri de căderi ale serverelor. O astfel de modelare constituie un prim pas pentru o abordare sistematică a posibilităţilor soft de transformare automată a unor căderi grave în căderi mult mai puţin grave la nivelul efectelor execuţiilor, activitate care ar asigura practic toleranţa la erori ale acelor sisteme de calcul distribuite.

1. Introduction. One of the original goals of building distributed systems was to make them more *reliable* than single processors systems, That is, if some machine goes down, some other machine takes over the job. There are various aspects regarding this concept.

Availability refers to the fraction of time in which the system is usable. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another way for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the task.

A main aspect related to reliability is *fault tolerance* [Tan92]. That is, what happens

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

when a server crashes and then quickly reboots ? In general, distributed systems can be designed to mask failures. If a system service is actually constructed from a group of closely cooperating servers, then it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. The challenge is to arrange this cooperation such as not to add substantial overhead to the system in the normal case, when everything is functioning correctly.

2. Types of failures. Distributed computing systems give algorithm designers the ability to write fault-tolerant applications in which correctly functioning processors can complete a computation despite the failure of others. It is well established that th complexity of writing such applications depends upon the type of faulty behaviour that processors may exhibit. For example, while simple stopping failures are relatively easy to tolerate, tolerating completely arbitrary behaviour can be much more difficult. To assist the designers of such applications, *translations* were developed that automatically convert algorithms tolerant of relatively benign types of failure into ones that tolerate more severe faulty behaviour [Bazzi93]. We give below a hierarchy of the most commonly considered failures, from the most easy ones to the most severe:

i). *crash failures* - in which the incriminated processors fail by stopping prematurely. Before they stop they behave correctly and after they stop they take no further actions.

ii). *send-omission failures* - the processor fails by intermitently omitting to send some of the messages that it should send, but the sent messages are always correct. Because these processors can fail and yet continue to send messages, their failure is more

difficult to detect and deal with than crash failures.

iii). *general omission failures* - the processor may stop or it may intermittently fail to send or receive messages [Perry86] and the sent messages are always correct. The identity of faulty processors is uncertain: did the sender or the receiver of an omitted message fail ?

iv). *arbitrary failures* - processors subject to arbitrary failures can take any action [Lamport82]. They can stop, omit to send messages, send spurious messages and falsely claim to have received messages they did not actually receive.

3. Protocols, histories and problem specifications.

Definition. A *distributed system* is a set D of n processors joined by bidirectional communication links. Processors do not share any memory, the communications being made through message passing. Each processor has a local *state* and we denote by Q the set of local states.

Processors communicate with each other in synchronous rounds. In each round, a processor first sends messages, then receives messages and then change its state. Let M be the set of messages that may be sent in the system and let $\square \notin M$ be the value that indicates "no message" and let $M'=M\cup\{\square\}$. Thus, if p sends no message to q in a round, we can say that p sends $\square$ to q, although no message is actually sent.

Definition. Processors run a *protocol* P, which specifies the messages to be sent and the state transitions. A protocol consists of two functions, a *message function* and a *state-transition function*. The message function is defined as $mf_p: \mathbb{N}\times D\times Q \rightarrow M$, where $\mathbb{N}$ is the set of positive integers. If processor p begins round i in state s, then P specifies that it

send $mf_p(i,p,s)$ to all processors in that round. The state-transition function is $st_p: \mathbf{N} \times D \times (M')^* \rightarrow Q$. If in round i processor p receives the messages $m_1,...,m_n$ from processors $p_1,...,p_n$ respectively, then P specifies that it change its state to $st_p(i,p,m_1,...,m_n)$ at the end of round i.

The code below illustrates the execution of a protocol P:

```
state := initial state;
for i=1 to ∞ do
        message := mf_p(i,p,state);
        if message ≠ □ then send message to all processors;
        foreach q∈D
                if received some m from q then get[q]:=m
                                          else get[q]:=□;
        state := st_p(i,p,get);
```

This definition of protocols appears restrictive in a sense. For example, every processor is required to broadcast a message in every round. A protocol's state transition function depends only on the messages that it just received and not on its previous state. Furthermore, processors are required to run forever and never halt. These restrictions were made for simplifying the presentation and they do not restrict the applicability of the results.

Histories describe the executions of a distributed system. Each history is a 4-tuple including the following elements: the protocol being run by the processors, the sequence of states through which the processors pass, the messages that the processors send and the messages that the processors receive. Formally, a history consists of a protocol and three functions. The functions define the states through which the processors pass and the messages sent and received by the processors in each round. A *state-sequence function* $sseq: \mathbf{N} \times D \rightarrow Q$ identifies the states of processors at the beginning of each round. *sseq(i,p)* is

the state in which processor p begins round i. A *message-sending function* msf:$\mathbf{N}\times D\times D$-->$M'$ identifies the messages sent in each round. *msf(i,p,q)* is the message that p sends to q in round i or □ if p sends no message to q in round i. A *message-receiving function* mrf:$\mathbf{N}\times D\times D$-->$M'$ identifies the messages received in each round. *mrf(i,p,q)* is the message that p receives from q in round i or □ if p does not receive a message from q in round i. Let mrf(i,p) be an abbreviation for the sequence mrf(i,p,1),..., mrf(i,p,n). H=(P,sseq,msf,mrf) is then a *history of protocol P*.

A *system* is identified with the set of all histories (of all protocols) in that system. A system can also be defined by giving the properties that its histories must satisfy. If S is a system and H=(P,sseq,msf,mrf)$\in S$, then H is a history of P running in S.

Protocols are run to solve particular problems. Formally, such problems can be specified by predicates on histories. Such a predicate, called a *specification*, distinguishes histories that solve the problem from those that do not.

Protocol P *solves problem with specification* Σ *(or solves* Σ*) in system* S if all histories of P running in S satisfy Σ. That is $\forall H\in S$ [H is of the form (P,sseq,msf,mrf) $\Rightarrow$ H satisfies Σ].

4. Correctness and failures. A processor executes *correctly* if its actions are always those specified by its protocol. Considering a history H = (P,sseq,msf,mrf), processor p *sends correctly in round* i *of* H if

$$\forall q\in D\ [msf(i,p,q) = mf_p(i,p,sseq(i,p))].$$

Processor p *receives correctly in round* i *of* H if

$$\forall q\in D\ [mrf(i,p,q) = msf(i,p,q)].$$

Processor p *makes a correct state transition in round* i *of* H if

$$sseq(i+1,p) = st_p(i,p,mrf(i,p)).$$

Processor *p is correct through round i of* H if it sends and receives correctly, and makes correct state transitions up to and including round *i* of H. Let

$$Correct(H,i) = \{p \in D \mid p \text{ is correct through round } i \text{ of } H\}.$$

We assume that all processors are initially correct, so $Correct(H,0)=D$. Then let $Correct(H)$, the set of all processors *correct throughout history* H, be $\cap_{i \in N} Correct(H,i)$. If a processor is not correct, it is *faulty*. Formally,

$$Faulty(H,i) = D - Correct(H,i) \text{ and}$$

$$Faulty(H) = D - Correct(H).$$

The following examples of formal specifications illustrate these definitions: Σ_1 specifies that "in round 7 processor p does not send correctly to q":

$$\Sigma_1 = \Sigma_1(P,sseq,msf,mrf) = msf(7,p,q) \neq mf_p(7,p,sseq(7,p)).$$

Σ_2 specifies that through round 10 at least 6 processors are correct:

$$\Sigma_2(H) = |Correct(H,10)| \geq 6.$$

Informally, a specification Σ is a state specification if it depends only on the state-sequence function and, in a certain way, on the set of correct processors. Formally, Σ is a state specification if

$$\forall H_1,H_2\ [(\Sigma(H_1) \wedge sseq_1=sseq_2 \wedge Correct(H_2) \subseteq Correct(H_1)) \Rightarrow \Sigma(H_2)].$$

Informally, a state specification Σ is failure-insensitive if it does not depend on the states of the faulty processors. Formally,

this means that

$$\forall H_1,H_2\ [(\Sigma(H_1) \wedge \forall i \in N\ \forall p \in Correct(H_2)[sseq_1(i,p)=sseq_2(i,p)]) \Rightarrow \Sigma(H_2)].$$

Individual processors may exhibit *failures*, that is to deviate from *correct* behaviour.

They may do so by failing to send or receive messages correctly or by otherwise not following their protocol. In the following we will formally define crash, send-omission, general omission, and arbitrary failures.

4.1. Crash Failures. A *crash failure* [Hadzilacos83] is the most simple type of failure that can appear. A processor commits a *crash failure* by prematurely halting in some round. Formally, p commits a crash failure in round $i_c \in \mathbf{N}$ of $H=(P,sseq,msf,mrf)$ if i_c is the least i such that $p \in Faulty(H,i)$ and if:

- p sends to each processor q either what the protocol specifies, or nothing at all:

$$\forall q \in D\ [msf(i_c,p,q) = mf_p(i_c,p,sseq(i_c,p)) \vee msf(i_c,p,q) = \Box];$$

and, afterwards,

- it sends no messages: $\forall i > i_c\ \forall q \in D\ [msf(i,p,q) = \Box]$,
- it receives no messages: $\forall i \geq i_c\ \forall q \in D\ [mrf(i,p,q) = \Box]$,
- it makes no state transitions: $\forall i > i_c\ [sseq(i,p) = sseq(i_c,p)]$.

The system $C(n,t)$ corresponds to the set of histories in which up to t processors commit only crash failures and all other processors are correct. That is, $H \in C(n,t)$ if and only if D can be partitioned into sets C and F such that $C=Correct(H)$, $|F| \leq t$, and

$$\forall p \in F\ \exists i_c \in \mathbf{N}\ [p \text{ commits a crash failure in round } i_c \text{ of } H].$$

4.2. Send-omission Failures. Another type of failure, called a *send-omission failure*, occurs if a processor omits to send messages [Hadzilacos84]. Processor p may commit such failures in history $H=(P,sseq,msf,mrf)$ if it always makes correct state transitions, receives correctly, and sends to each processor what its protocol specifies or

nothing at all:

$$\forall i \in \mathbf{N}\ \forall q \in D\ [msf(i,p,q) = mf_p(i,p,sseq(i,p)) \vee msf(i,p,q) = \Box];$$

The system $S(n,t)$ corresponds to the set of histories in which up to t processors are subject to send-omission failures and all other processors are correct.

While crash failures are relatively easy to tolerate, intermittent send-omission failures are more difficult to identify and compensate. If processors may omit to send messages and later function correctly, then the correct processors may have more difficulty agreeing on the identity and timing of failures than they would if only crash failures occured.

4.3. General Omission Failures. A more complex type of failure, called a *general omission failure* [Perry86], occurs if a processor intermittently fails to send and receive messages. Processor p may commit such failures in history H=(P,sseq,msf,mrf) if it always makes correct state transitions, always sends to each processor what its protocol specifies or nothing at all, and always receives what was sent to it or nothing at all:

$$\forall i \in \mathbf{N}\ \forall q \in D\ [msf(i,p,q) = mf_p(i,p,sseq(i,p)) \vee msf(i,p,q) = \Box];$$

$$\forall i \in \mathbf{N}\ \forall q \in D\ [mrf(i,p,q) = msf(i,q,p) \vee mrf(i,p,q) = \Box].$$

The system $G(n,t)$ corresponds to the set of histories in which up to t processors are subject to general omission failures and all other processors are correct.

General omission failures are more difficult to tolerate than send-omission failures. In addition to the uncertainty regarding the timing of failures, there may also be uncertainty as to the identify of the faulty processors: if an omitted message is detected, it may be difficult to tell whether it is the sender or the receiver that is at fault. Furthermore,

faulty processors may be sending incomplete information, as they may have omitted to receive message from correct processors in previous rounds.

4.4. Arbitrary Failures. Crash failures considerably restrict the behaviour of faulty processors. Omission failures place fewer restrictions on this behaviour. In the worst case, faulty behaviour may be completely arbitrary. Processors may fail by sending incorrect messages and by making arbitrary state transitions [Lamport82]. Processor p *is subject to arbitrary failures* in history H=(P,sseq,msf,mrf) if it may deviate from P in any way. It may do one or more of the following:

- fail to send correctly: $\exists i \in \mathbf{N}\ \exists q \in D$ [msf(i,p,q) ≠ mf_p(i,p,sseq(i,p))],
- fail to receive correctly: $\exists i \in \mathbf{N}\ \exists q \in D$ [mrf(i,p,q) ≠ ms(i,q,p)], or
- make an incorrect state transition: $\exists i \in \mathbf{N}$ [sseq(i+1,p) ≠ st_p(i,p,mrf(i,p))].

The system A(n,t) corresponds to the set of histories in which up to t processors commit arbitrary failures and all other processors are correct. It is clear that arbitrary failures are more difficult to tolerate than the other kinds. Faulty processors may actively try to confuse the correct ones, they being able even to "cooperate" to make fault-tolerance even more difficult to achieve.

5. Conclusions. As distributed systems become more and more widespread, the demand for fault tolerance is one of the main request from a distributed computing system. Such systems will need considerable redundancy in hardware and the communication infrastructure, but they will also need it in software and data.

To achieve the goal of a fault tolerant distributed system we have first to

distinguish between the types of failures a system may exhibit and try to model these failures and the behaviour of the system in the presence of these failures. The ultimate goal, based on the ideas from [Bazzi93], will be to develop translations from one type of failure to another, translations destinated to ease the system tolerance to failures. The model presented here may be a basis for developing such a scheme of translations.

REFERENCES

[Bazzi93] R.Bazzi, G.Neiger - Simplifying Fault-Tolerance: Providing the Abstraction of Crash Failures, *Technical Report GIT-CC-93/12*, Georgia Institute of Technology, 1993.

[Hadzilacos83] V.Hadzilacos - Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies), *Technical Report 18-83, Aiken Computation Laboratory, Harvard University, 1983, Ph.D. disertation.*

[Hadzilacos84] V.Hadzilacos - Issues of Fault Tolerance in Concurrent Computations , *Technical Report 11-84, Aiken Computation Laboratory, Harvard University, 1984, Ph.D. disertation.*

[Lamport82] L.Lamport, R.Shostak, M.Pease - The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems*, 4(3), pp.382-401, July 1982.

[Perry86] K.J.Perry, S.Toueg - Distributed agreement in the presence of processor and communication faults, *IEEE Transactions on Software Engineering*, 12(3), pp.477-482, March 1986.

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

PROFESSOR SEVER GROZE AT HIS 65TH ANNIVERSARY

by

Gh. Coman* and M. Frențiu*

Professor S. Groze was born on November 29, 1929 in Telciu, Bistrița-Năsăud. After finishing secondary school in 1948, he studied at the University of Cluj. In 1952, after his graduation, he was appointed as assistant at the Department of Mathematics, University of Cluj. From 1960 he worked as an assistant professor at the Pedagogical Institut of Baia-Mare. In 1972 he returned at the University of Cluj-Napoca. In 1980 he became full professor of this Faculty. In all this period, Professor S. Groze gave many courses and seminars on algebra, analysis, geometry, numerical analysis, computer science, etc. His courses were held at a high scientific and pedagogical level. So, he is remarked as an eminent mathematician and a distingused pedagogue, at the same time, with true love and devotion for his students.

Simultaneously with his pedagogical work, Professor Groze has developed an appreciated research work in geometry, numerical analysis and computer science. But, the prefered topic is numerical analysis in which he received his Ph. D. degree in 1971. In these topics he has written a lot of books for students and research papers.

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

Professor S. Groze is also a very good organizer. So for several years he served as Vice-Rector of the Pedagogical Institut of Cluj (1962-1964), Dean of the Faculty of Mathematics, Physics and Chemistry of the same institut (1964-1966, 1968-1972), Rector of the Pedagogical Institut of Baia-Mare (1966-1968), Vice-Dean of the Faculty of Mathematics, University of Cluj-Napoca (1977-1981), etc.

Also, Professor S. Groze was the head of the Computer Science Group of the Faculty of Mathematics and Physics.

Professor S. Groze is never less than generous to his collaborators and working with him is, first of all, a great pleasure.

Now, on celebrating his 65-th birthday, we wish him Many Happy Return of the Day and a long life in health and happiness

LIST OF THE SCIENTIFICAL WORK

of Professor Sever Groze

1. Despre o generalizare a contactelor nomogramelor cu transparent. Studia Univ.Babeş-Bolyai, Series I, Fasc.1, Math.-Physica, 1961, pp. 169-174.
2. Sur un classe de transformations des nomogrames de l'order trois, Mathematica vol.7, (30), 2, 1965,pp. 233-246.
3. La décomposition d'une projectivité sur une conique et son application à la meilleure transformation projective d'une echelle situé sur un circle. Revue Roumaine des Mathem. pures et appliq., Tom XII, nr.8,1967, pp.1065-1073.

4. Sur la transformation projective d'un nomograme ayant deux echelles sur un circle et la troisième sur une courbe quelconque. Revue Roumaine des Math pures et appl., nr. 2, Tom XV, 1970, pp. 245-254.

5. Criterii pentru ca o nomogramă cu puncte aliniate să aibă eroare minimă. Studia Univ. Babeş-Bolyai, Fasc.1, 1970, pp.69-73.

6. Metoda lui Steffensen aplicată la rezolvarea ecuaţiilor operaţionale neliniare definite în spaţii supermetrice. Studii şi Cerc. mat., 5, Tom.23, 1971, pp.711-717.

7. Asupra metodei coardei în rezolvarea ecuaţiilor operaţionale definite în spaţii supermetrice. Studii şi Cerc.mat. 5, Tom. 23, 1971, pp. 719-725.

8. Asupra rezolvării ecuaţiilor operaţionale definite în spaţii L-supermetrice. Studia Univ.Babeş-Bolyai, Series Math.- Mech., Fasc.1, 1971, pp. 81-85.

9. Asupra diferenţelor divizate generalizate. Anal. Şt. ale Univ. "Al.I.Cuza" Iaşi,(Serie Nouă), Sectiunea I, Matematica, Tom XVII, 1971, Fasc.2, pp.375-379.

10. Principiul majorantei şi rezolvarea ecuaţiilor operaţionale neliniare definite în spaţii supermetrice prin metoda aproximaţiilor succesive. Anal.St.ale Univ. "A.I.Cuza", Iasi,(Serie Noua), Secţiunea I, Matematică, Tom XVIII, Fas.1, pp.75-79.

11. Sur la methode de Newton dans les espaces L-supermetriques. Studia Univ. Babeş-Bolyai, Series Math.- Mech.,Fasc.1,1972, pp. 55-59.

12. Asupra condiţiilor de convergenţă la metoda coardei în spaţii supermetrice Studia Univ. Babeş-Bolyai, Series Math.-Mech., Fasc.1, 1973, pp.55-59.

13. Asupra rezolvării ecuaţiilor operaţionale neliniare printr-o metodă analogă cu a hiperbolelor tangente. Studia Univ. Babeş-Bolyai, Serie Math.- Mech., Fasc.2, 1973, pp.47-50.

14. Principiul majorantei în rezolvarea ecuaţiilor operaţionale neliniare. Studia Univ. Babeş-Bolyai, Serie. Math.-Mech., Fasc.1,1974,pp.69-74.

15. Principiul majorantei şi metoda coardei. Stud. Univ.Babeş-Bolyai, Mathematica, 4, 1975, pp.50-54.

16. Application of Iterative Methods for Solving Operator Equations and Improving Convergence Conditions. Revue d'Analyse numerique et de theorie de l'approximation. Tom.6, Nr.1, 1977, pp. 15-21.

17. Metode convergente de ordinul k în spaţii supermetrice. Studia Univ. Babeş-Bolyai, Mathematica, 2, 1977, pp.23-28.

18. The Method of Chords for solving Operator Equations dependent on one Parameter. Revue d'Analyse numerique et de l'approximation. Tom 8, nr.2, 1979, pp.181-185.

19. The Principle of the Majorant in Solving Operator Equations which depend on Parameter. Revue d'Analyse numerique et d'Approximation. Tom 8, Nr.2, 1977, pp.177-180.

20. On Steffensen's Method for Solving Nonlinear Operator Equations defined in Frechet Spaces. Studia Univ. Babeş-Bolyai, Mathematica, XXV, 1, 1980, pp.62-66.

21. Rezolvarea ecuaţiilor operaţionale neliniare în spaţii Frechet printr-o metodă analogă cu a parabolelor tangente. Studia Univ.Babeş-Bolyai, Mathematica, XXVI, 3, 1981, pp.24-28.

22. A Method analoguos to the Chebyshev Method for Solving of the Operator Equations. Studia Univ.Babeş-Bolyai, Mathematica, XXVI, 2, 1981, pp.72-75.

23. Succesive Approximations in Uniform Spaces. Proceedings of the Colloquim on Approxim. and Optim., Cluj-Napoca, 1984, pp.105-110.

24. A Combined Iterative Method in Fréchet Spaces. Itinerant Seminar on Functional Equat., Approximation and Convexity, Cluj-Napoca, 1985. pp.79-85.

25. Steffensen's Method in Fréchet Spaces. Research Seminars, Preprint Nr.4, 1985, pp.68-75.

26. A Method Analoguous to the Method of the Tangent Hyperbolas in Fréchet Spaces. Lucrările Simpozionului "Informatica şi aplicaţiile sale", Cluj-Napoca, 1985, pp.78-83.

27. On modified Method of Chords for Operator, Equations whith Parameters and Applications. Research Seminars, Preprint Nr.2, 1986, pp.29-36.

28. An Algorithm Corresponding to the Method of Chords in Fréchet Spaces. Studia Univ. Babeş-Bolyai, Mathematica, XXVII, 3, 1987, pp.37-45.

29. A Method Analoguos to the Chebyshev Method for the Solving of the Operator Equations Definited in Fréchet Spaces, Studia Univ. Babeş-Bolyai, Mathematica, XXXII, 4, 1987, pp.33-36.

30. On the Solving of Operatorial Equations Defined in Fréchet Space by a Modified Chord Method. Studia Univ.Babeş-Bolyai, Mathematica, XXXII, 1987, pp.37-40.

31. A Class of Iterative Method in Fréchet Spaces. Research Seminars, Preprint nr.5, 1987, pp.28-36.

32. On the Steffensen's Method in Frechet Spaces. Research Seminars, Preprint nr.9, 1987, pp.134-142.

33. On a Method Analoguous to Steffensen's Method. Research Seminars, Preprint nr.9, 1988, pp.17-26

34. The Principle of the Majorant in Solving on Nonlinear Operatorial Equations in

Frechet Spaces. Res.Seminars, Preprint nr. 9, 1988, pp.27-32.

35. On the Convergence of a Method Analoguous to Method of Tangent Hyperbolas in Frechet Spaces. Research Seminars, Preprint, nr.9, 1989, pp.41-50.

36. On the Method Convergente of k-order in Frechet Spaces. Research Seminars, Preprint nr.2, 1989, pp.50-67.

37. On the Convergence of Class of Iterative Method in Frechet Spaces, Revue d'Analyse numerique, Vol.XIX, 1990, pp.45-49.

38. The Principle of the Majorant and the Method Analoguous to the Chebyshev Method for the Solving Operatorial Equations which Depend on Parameters. Studia Univ. Babeş-Bolyai,Mathematica,Nr.3,1990,pp.45-50.

A Combined Iterative Method for Solving Operator Equations in Frechet Spaces. Studia Univ.Babeş-Bolyai,Mathematica,Nr. 3,1990, pp.25-30.

40. The Principle of the Majorant in Solving of Nonlinear Operatorial Equations which Depends on One Parameter, Defined in Frechet Spaces. Bul.Şt.al Univ. Baia-Mare, seria B.Vol.VIII, 1991, pp.89-96.

On the Convergente of Three Order Method in Frechet Spaces, Studia Univ. Babeş-Bolyai,Mathematica,XXXVI,pp,34-39

The Principle of the Majorant in Solving of an Operatorial Equation Defined in Frechet Spaces by a Convergence of the Three Order Method. Research Seminars, Preprint no.5, 1992, pp.125-132.

43. Consequences of Theorems Concerning the Convergence of Chord Method in Frechet Spaces. Studia Univ. Babeş-Bolyai, Mathematica, XXXVIII, 1993, pp.59-64.

On the Chord Method in Frechet Spaces. În curs de apariţie, Studia Univ. Babeş-Bolyai, Mathematica.

BOOKS AND UNIVERSITY LECTURES

1. Aritmetica. Litografia Univ.Cluj-Napoca,1971.
2. Bazele informaticii. Litografia Univ.Cluj-Napoca,1977 (în colaborare).
3. Bazele informaticii. Culegere de probleme. Litografia Univ. 1976 (în colaborare).
4. Rezolvarea numerică a ecuaţiilor operaţionale. Litografia Univ.1981 (în colaborare).
5. Bazele informaticii. Culegere de probleme pentru laborator. Litografia Univ.1982. (în colaborare).
6. Programare şi informatica. Litografia Univ.1982 (în colaborare).
7. Bazele informaticii I (Ed.1). Litografia Univ.1983 (în colaborare).
8. Bazele informaticii I (Ed.2). Litografia Univ.1986 (în colaborare).
9. MATH-1, Scientific Program Library for Roumanian Personal Computer, Cluj-Napoca,1987 (în colaborare).
10. Elemente de informatică pentru licee. Litografia Univ.1988 (în colaborare)
11. Bazele informaticii. Limbajele BASIC şi PASCAL. Litografia Univ. Babeş-Bolyai,1992. (în colaborare).
12. Bazele informaticii I. Lit.Univ.D.Cantemir, Cluj-Napoca 1992 (în colaborare)
13. Elemente de informatică pentru licee (Ed.2). Editura "Microinformatica" 1992. (În colaborare).
14. Elemente de informatică pentru licee (Ed.3). Editura "Microinformatica", 1993 (În colaborare).
15. Bazele informaticii - Culegere de probleme. Lit.Univ.Babeş-Bolyai, 1993

STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

(în colaborare).

16. **Pascal Programming. Illustrative Examples and Proposed Problems.** Editura Promedia, 229 pages, 1995.

17. **Pascal Programming for Schools.** Editura Microinformatica, 1995.

în următoarele serii:

matematică (trimestrial)
fizică (semestrial)
chimie (semestrial)
geologie (semestrial)
geografie (semestrial)
biologie (semestrial)
filosofie (semestrial)
sociologie-politologie (semestrial)
psihologie-pedagogie (semestrial)
ştiinţe economice (semestrial)
ştiinţe juridice (semestrial)
istorie (semestrial)
filologie (trimestrial)
teologie ortodoxă (semestrial)
educaţie fizică (semestrial)

In the XXXIX-th year of its publication (1994) *Studia Universitatis Babeş-Bolyai* is issued in the following series:

mathematics (quarterly)
physics (semesterily)
chemistry (semesterily)
geology (semesterily)
geography (semesterily)
biology (semesterily)
philosophy (semesterily)
sociology-politology (semesterily)
psychology-pedagogy (semesterily)
economic sciences (semesterily)
juridical sciences (semesterily)
history (semesterily)
philology (quarterly)
orthodox theologie (semesterily)
physical training (semesterily)

Dans sa XXXIX-e année (1994) *Studia Universitatis Babeş-Bolyai* paraît dans les séries suivantes:

mathématiques (trimestriellement)
physique (semestriellement)
chimie (semestriellement)
geologie (semestriellement)
géographie (semestriellement)
biologie (semestriellement)
philosophie (semestriellement)
sociologie-politologie (semestriellement)
psychologie-pédagogie (semestriellement)
sciences économiques (semestriellement)
sciences juridiques (semestriellement)
histoire (semestriellement)
philologie (trimestriellement)
théologie orthodoxe (semestriellement)
éducation physique (semestriellement)