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.

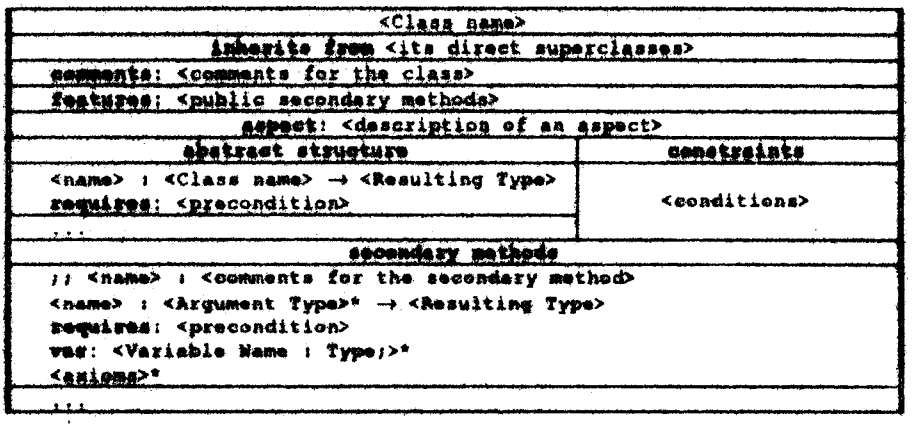


Figure 1: The generic box for a class

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:

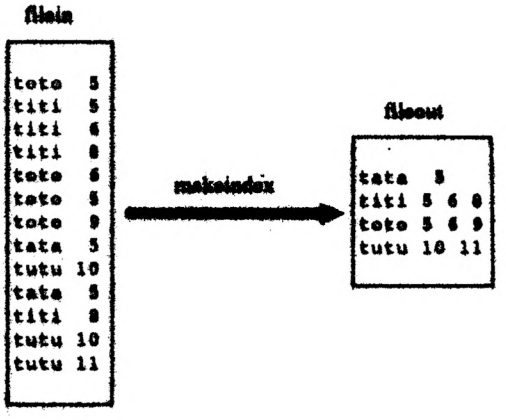


Figure 2: a `\makeindex` example

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.

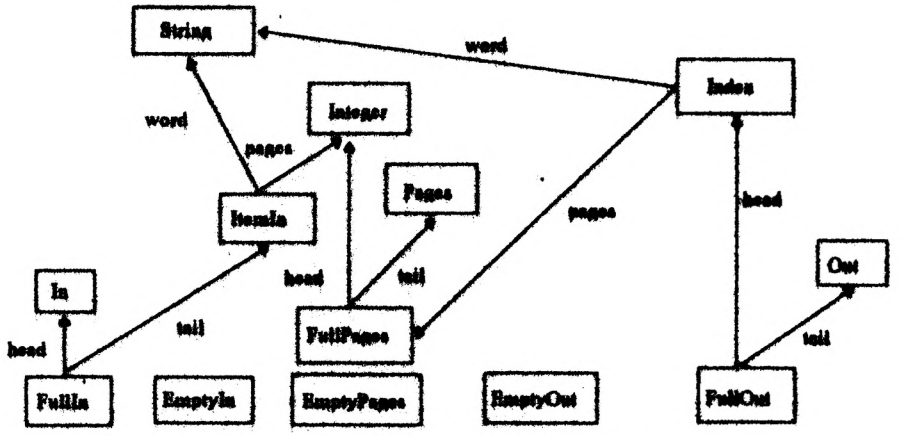


Figure 6: The Structural Dependency Graph

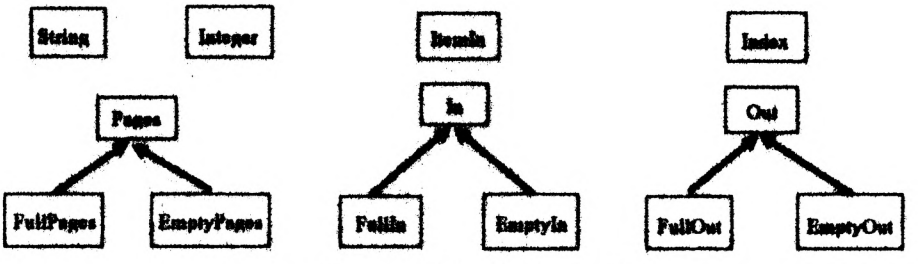


Figure 7: A part of the Inheritance Graph

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.

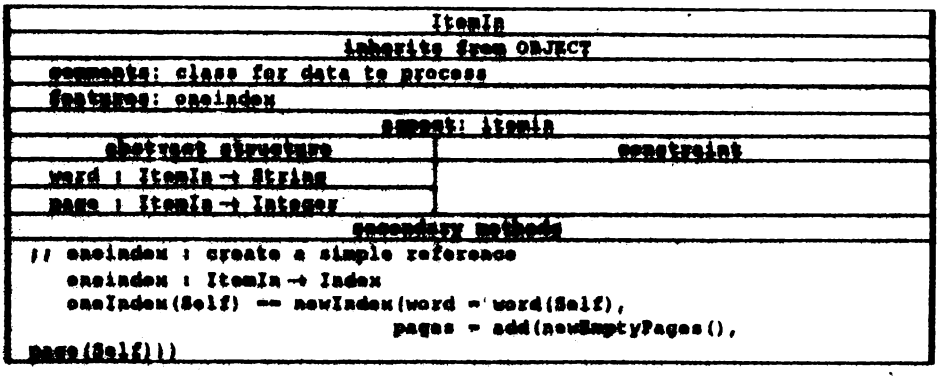
Some extensions are under study: metaclasses (based on the ObjVLisp model), schemes, methods as objects and association types. These extensions add difficulties to type checking.

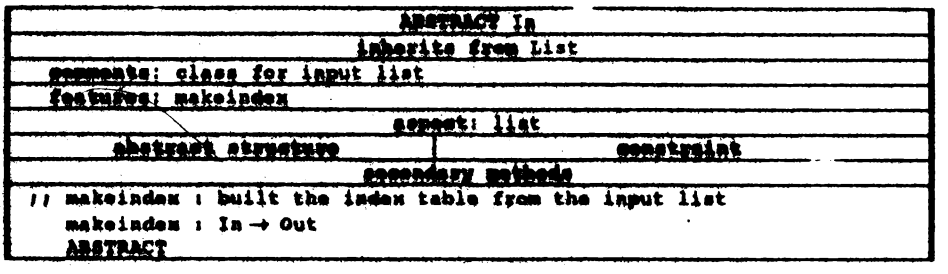
The main features of our model are:

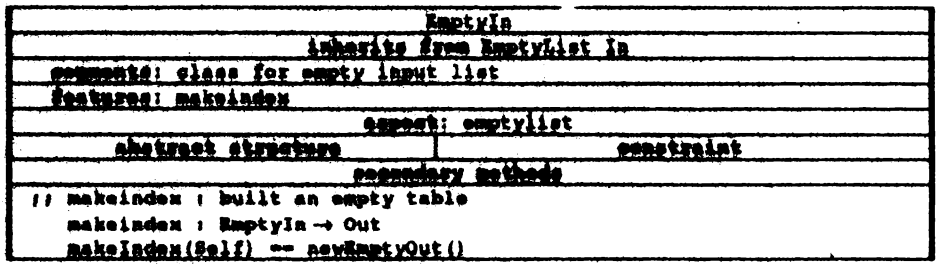
- an object-oriented and formal model to abstractly design applications, i.e. without the need of a particular Object-Oriented Language,
- rules and criterion to check graphs, types, method redefinitions, inheritance links, ...
- a symbolic evaluator and proof technique,
- a direct implementation in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

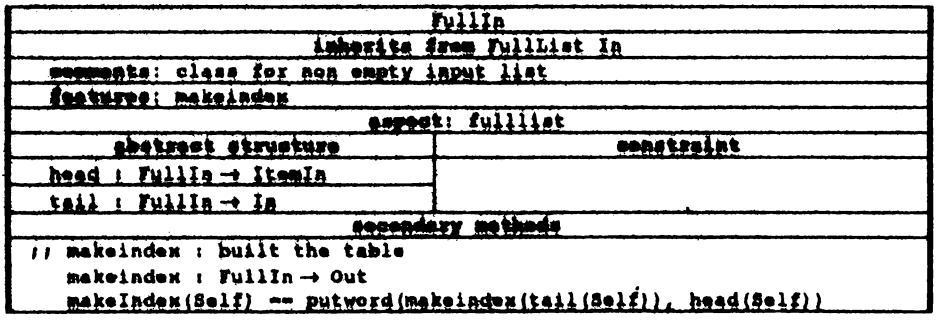
APPENDIX

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