The Formal Class Model

Pascal André[1], Dan Chiorean[2] and Jean-Claude Royer[1]

September 2, 1996

[1] ERTO-IRIN, Faculté des Sciences et des Techniques, Université de Nantes, 44072 Nantes Cédex 03, FRANCE, Tel: (33) 40 37 30 22 - Fax: (33) 40 09 32 51, e-mail: (andre,royer)@unantes.univ-nantes.fr
[2] Laboratorul de Cercetare în Informatică, Facultatea de Matematică şi Informatică, Universitatea "BABES-BOLYAI", 3400 Cluj-Napoca, ROMANIA, Tel: (40) 64 11 29 62 - Fax (40) 64 11 19 05

Abstract

This paper describes the main features of an object-oriented model with classes, inheritance and strong typing. This model is a functional model to abstractly design object-oriented applications. We will explain some basic rules about class construction, inheritance, natural method redefinition and subtyping. A symbolic evaluator based on term rewriting and a concrete translation into the Eiffel language are presented.
Keyword Codes: D.1.5; D.2.1; D.3.3
Keywords: Programming Techniques: Object-Oriented Programming; Software Engineering: Requirements/Specifications; Programming Languages: Language Constructs and Features

1 Introduction

Object-orientation is a good approach to the development of large systems. Modularity, abstraction, encapsulation and inheritance are fruitful concepts to analyse and design software systems. The main advantages of object orientation are reusability, extensibility and flexibility.

Object-oriented analysis and design require formal specifications to make proofs, verifications and automatic processing possible. Formal specifications increase the software qualities of class and enable to design consistent and complete libraries of classes. A formal and abstract model to specify classes is also useful to facilitate reverse engineering or application rewriting. Such a model is essential to compare or to reuse classes either in a language or between various languages. A last benefit from it is that one can teach object-oriented programming (OOP) in a more abstract way than with concrete object-oriented languages.

Current widespread formal approaches to system specification using object-oriented techniques are often based on model-oriented specification and mathematical languages like Z or VDM (Object-Z, VDM++, Z++ [DD90, DVK92, LH92]). Often they are too operational and have the drawback of having a lack of animation tools. We prefer to see object-oriented design as: *" the construction of software systems as structured collections of abstract data type implementations"* [Mey88].

We define a model with the notion of formal class. A formal class is an abstraction of concrete class in languages such as C++, Eiffel [Mey88], CLOS [BDG$^+$88] or Smalltalk. A formal class is an algebraic specification (like abstract data type is) with an object-orientation. This general model is functional and unifies the major concepts of OOP. It can be used to build formal specifications or to design a system. It is also a laboratory to experiment constructions, study object-oriented concepts and design rules.

This paper is organized as follows: first it presents broad ideas and syntax of the model in section 2. Section 3 details the definition of classes, aspect construction and some method writing. Section 4 presents multiple inheritance and section 5 the safe type checking. Section 6 gives an abstract operational semantic using rewriting techniques and a class translation in Eiffel language. Finally a comparison with other works is given.

2 A Specification and Design Model

Object-oriented programming is useful for software engineering for several reasons, the main reason is software design of reusable components. There are three main cases of class reuse: inheritance, structuration and genericity. Here we are concerned with languages with classes and multiple inheritance. We consider that a class defines a type and inheritance implies subtyping. Furthermore our purpose is to use such languages to formally design systems. We present in this section general concepts and notations of the object model. The model corresponds to modular design where formal classes are modules. The model uses strict rules and principles for class design, inheritance and typing. In real applications it is sometimes necessary to make these constraints more flexible, but this can be done at implementation level rather than at specification level. One knows that starting from a correct design and refining it step by step is a good approach [Jon86].

2.1 Class Design

Our conception of object is based on the notion of abstract description (i.e. a tuple of informations that characterises this object). A class defines an **aspect** allowing abstract description of its instances. An aspect defines a family of class implementations rather than a single implementation as instances variables, slots or fields in concrete languages. Once the aspect design is made the class is well defined and its behaviour can be incrementally extended or the class can be reused. The property of incremental extension for the behaviour is an important advantage in software engineering.

A class also defines the instance behaviour. From the user point of view the behaviour of a class is split in two parts: **primitive methods** and extension methods (**extensions** to be short).

The primitive methods are associated with the aspect and allow to create, to copy, to compare and to describe objects. Primitive methods characterise the behaviour subset defined by the following property: removing a primitive method modifies:

- the set of described instances,
- or does not allow access to information of instances,
- or does not make comparison or description of instances possible.

Other methods are called extensions because their semantics is based on primitive methods. Often extensions are functional extensions of primitive methods, i.e. every application of an extension can be reduced to applications of primitive methods. The notion of primitive and extensions are usual in specification of abstract data types [Hor78, GL87, Lin89].

2.2 Formal Classes

A class presentation is a "box". The information provided about a class consists of its name, exported and imported features, inheritance links, aspect and extensions.

<Class Name>	
inherits from <Direct Superclass Names>	
comments: <String: class comment>	
features: <Public Method Names>	
from: <Type Name> **import:** <Operation Name>*	
aspect :	
field selectors	**constraint**
<Method Name> : <Class Name> $\longrightarrow$ <Resulting Type> **requires:** <Condition>	<Condition>
... : <Class Name> $\longrightarrow$ ...	...
extensions	
;; <Method Name> : <String: method comment> <Method Name> : <Argument Type Names> $\longrightarrow$ <Resulting Type Name> **var:** <Variable : Type;>* <axiom>*	
...	

Figure 1: General Presentation for Formal Class

There are two main parts: the aspect and the extensions. The design of a class begins with the definition of its aspect. An aspect of a class has two parts: an abstract structure and a constraint. The abstract structure is a set of field selectors. Each field selector is a method with profile: $\texttt{fsel}_i$: CFC $\longrightarrow$ $\texttt{T}_i$, where CFC means current formal class. The constraint is a predicate on this abstract structure. An extension in the FC (Formal Class) model is characterised by a profile, a precondition and axioms. An extension can be declared **ABSTRACT** so it has no axiom.

2.3 Abstract Syntax

To be more general, we assume the existence of predefined types and associated operations. Predefined type and class names begin with an upper case letter. Operations and methods are lower case names with underscore. Variables are denoted by `Self` or introduced by the keyword **var:**. We use terms, equations and axioms as in algebraic specifications or in functional languages.

```
Axiom       ::= Condition ==> Equation
Condition   ::= null | Equation ∧ Condition
Equation    ::= Term == Term
Term        ::= Variable | Application | Message
```

An axiom is a positive conditional equation. An equation is a pair of algebraic terms `t == u`. A term is either a variable or the application of an operation on terms or a message. Message send is written as a functional call:

`<selector>(<receiver> <, argument>*).`

Here we only use single selection (single dispatch) and as usual the receiver variable is denoted by `Self` in methods.

Each method contains at least the current class in its profile. We distinguish two main groups of operations (resp. methods), those having the current class as resulting type, called the **constructors** and those having another type as resulting type, called the **observers** (these terms are usual in ADT).

Instance methods and class methods are distinguished according to their profile. The profile of a class method is $\mathtt{m} \ : \ \mathtt{A}_1 \ \ldots \ \mathtt{A}_n \ \longrightarrow \mathtt{CFC}$ where its $\mathtt{A}_i$ are not equal to `CFC`. The profile of an instance method is $\mathtt{m} \ : \ \mathtt{CFC} \ \mathtt{A}_2 \ \ldots \ \mathtt{A}_n \ \longrightarrow \mathtt{A}_{n+1}$. Only instance methods are overloaded.

2.4 An example

To present the class construction we use an example. We want to design classes which represent the structure and the declarations of a program written in a language with lexical scope. We suppose predefined types or classes exist: `Boolean, Integer, String` and generic `List[T]`. Such definitions are possible with formal classes; [AR94] describes "a hierarchical design" of lists with formal classes, which allows genericity. An example of program structure is:

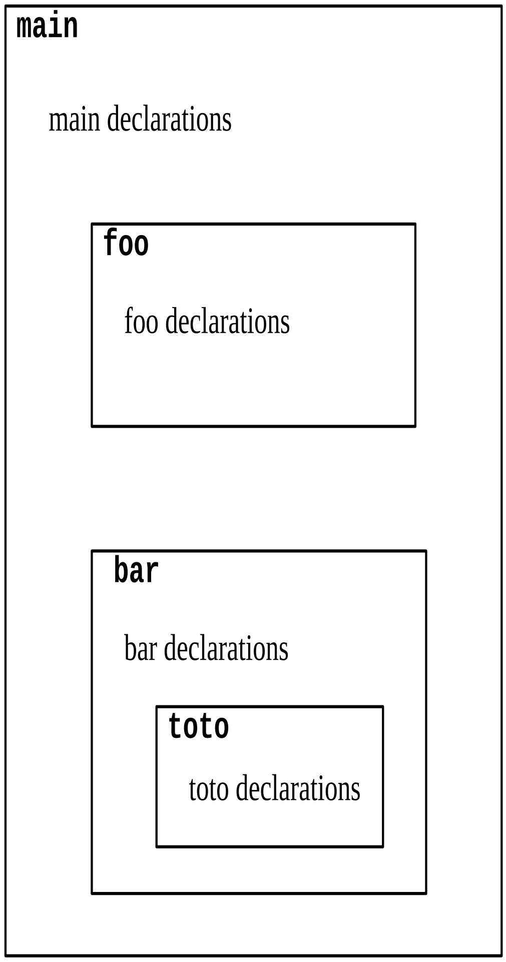


Figure 1: An Example of Block Structured Program

To design a solution to this problem we define three classes: `DecTable`, `Terminal` and `Node`. The `DecTable` class represents the general construction of blocks with a name and local declarations. Class `Terminal` represents a simple lexical block without inner blocks. The last class represents blocks with internal blocks.

ABSTRACT DecTable	
inherits from OBJECT	
comments: The abstract class for blocks	
features: number, all, valid?, environment, ...	
from: DecList **import:** length	
aspect : general block	
field selectors	**constraint**
name : ABSTRACT DecTable ⟶ String	
declarations : ABSTRACT DecTable ⟶ DecList	
extensions	
;; number : compute the number of local declarations number : DecTable ⟶ Integer number(Self) == length(declarations(Self))	
;; all : number of all the declarations all : DecTable ⟶ Integer ABSTRACT	
;; valid? : tests if the path is right in the program valid? : DecTable Path ⟶ Boolean ABSTRACT	
;; environment : compute the lexical environment environment : DecTable Path ⟶ DecList **requires:** valid?(Self) ABSTRACT	
...	

Figure 3: The Abstract Formal Class DecTable

A value of type DecList is a list of Declaration with the constraint that identifiers are single. A Declaration represents type, variable or constant declarations in the given language. The class Path is a specialization of List[String] which describes the path of a block in the program.

Field selectors of class DecTable are name and declarations, there is no constraint. We recall that field selectors are primitive methods. Other primitive methods are defined for all classes starting from the aspect: new<CFC> the primitive generator of class CFC, copy the recursive constructor, describe the external description and equal? the semantic equality are the most important.

Class DecTable is declared **ABSTRACT** it means that this class has no new<CFC> to create instances[1]. The above example describes a single extension and three abstract methods.

Primitive methods are implicitly public. Features are public extensions. Furthermore it is possible to use a **from: import:** clause as in Modula or CLU.

3 Building Formal Classes

This section describes the construction of formal classes.

[1]More precisely it means that this class has no generator.

3.1 Aspect

To design a class we first build its aspect. Remember that the field selectors are necessary and sufficient observers to describe class instances.

Terminal		
inherits from `DecTable`		
comments: `The class for simple blocks`		
features: `number, all, valid?, environment, ...`		
aspect : `simple block`		
field selectors		**constraint**
`name : Terminal`	$\longrightarrow$ `String`	
`declarations : Terminal`	$\longrightarrow$ `DecList`	
extensions		
`...`		

Figure 4: Aspect of Class `Terminal`

The resulting types of the field selectors are named the structuring types of the class and noted $\mathtt{T}_i$. Structural dependency[2] is the set of links between a formal class and its structuring types. This defines a relation between types with predefined types or formal classes as leaves and formal classes as nodes. Our `DecTable` class structurally depends on `String` and `DecList`.

Definition 3.1 (Well-Structured Class)
A well-structured class defines at least one instance and its instances are finitely generated[3].

This definition implies a well-found induction on instances and messages. If the fields selectors preconditions are equivalent to true then it exists a necessary and sufficient criterion.

Definition 3.2 (Simple Structural Checking)
If all field selector preconditions are equivalent to true then
the structural dependency graph of the class is without cycle $\Longleftrightarrow$ *the class is well-structured.*

This criterion is true for class `Terminal` in figure 4.

Let the `FullDecTable` class for list of `DecTable` with at least one `DecTable`. A naive and bad aspect design for this class is:

FullDecTable		
aspect :		
field selectors		**constraint**
`head : FullDecTable`	$\longrightarrow$ `DecTable`	...
`tail : FullDecTable`	$\longrightarrow$ `FullDecTable`	...

Figure 5: Bad Design

It structural dependency graph contains a cycle on `FullDecTable`.

A more general and necessary condition exists.

[2]It is an abstraction of the concrete notion of aggregation.

[3]Extension to infinite object is possible but needs call-by-name or lazy evaluation see footnote 4.

Definition 3.3 (Well-Structuring Criterion)
A class is well-structured then for every field selector:

- *either* $\mathtt{T}_i$ *does not structurally depend on CFC and* $\mathtt{T}_i$ *is well-structured or it is a predefined type.*
- *or* $\mathtt{T}_i$ *is a formal class and structurally depends on CFC and the field selector precondition is not equivalent to true.*

FullDecTable	
aspect :	
field selectors	**constraint**
`head : FullDecTable` $\longrightarrow$ `DecTable`	...
`single? : FullDecTable` $\longrightarrow$ `Boolean`	...
`tail : FullDecTable` $\longrightarrow$ `FullDecTable` **requires:** `single?(Self) == false`	...

Figure 6: A Flat and Right Design for Class `FullDecTable`

The previous criterion is true in the above "flat" design; in fact this is a well-structured class. In this model there are two main ways to design a class: "hierarchical" using inheritance or "flat" using structural dependency. At first sight the "hierarchical" design seems more complex but in general it has two benefits: a better reuse and a finer type checking. Questions about class reuse are out of the scope of this paper.

3.2 Primitive Methods

An instance is represented by an algebraic term like:

```
bloc = newTerminal(name= ‘‘toto’’, declarations= ...)
```

Keywords are allowed in some primitive methods to clarify messages. Examples of messages with primitive methods are:
`copy(bloc, name= ‘‘titi’’)`,
`describe(copy(bloc, name= ‘‘titi’’))`,
`equal?(copy(bloc, name= ‘‘titi’’), bloc)`,
`name(copy(bloc, name= ‘‘titi’’))`,
`declarations(copy(bloc, name= ‘‘titi’’))`. Note that object identity has no meaning since we have a functional model and then referential transparency.

Some class designs need the use of a constraint. For example to avoid ambiguities on paths, we need a non trivial constraint on the abstract structure of `FullDecTable` (figure 6). It is a conjunction of equations denoting a field selector predicate: `single?(Self) == true or (single?(Self) == false and`
`used?(tail(Self), name(head(Self))) == false)`.
This constraint means that the first name is not used anywhere at the same level in the rest of the list. Uner these conditions every block has a unique path in the program. The constraint can be seen as a condition for creating or modifying an object. When defining extensions the designer must preserve the constraint. This is an abstract idea of class invariant, as in Eiffel or CLU.

3.3 Extensions

We adapted design criteria from abstract data types ([Hor78, Mus80, Gau90]) then we get a notion of well-designed class.

Definition 3.4 (Well-Designed Class)
A well-designed class is: well-structured, consistent and fully specified.

We do not detail the formal definitions of consistency and of full specification. The idea of the consistency property is: a term is evaluated to a single value. Some criteria and algorithms, based on rewriting techniques, exists for consistency (see [JL86] for more details). From now on we assume that axioms are implicit conditional rewrite rules (from left to right: `Condition ==> Term --> Term`) and define a convergent system. The full specification property means that applying any observer to a correct object of the class must produce a value with the corresponding type. The full specification property can be obtained by some kind of axiom writing.

```
Terminal
-------------------------------------------------------------
extensions
-------------------------------------------------------------
;; all : number of all the declarations
   all : Terminal ⟶ Integer
all(Self) == number(Self)
-------------------------------------------------------------
;; valid? : tests if the path is a correct path in the program
   valid? : Terminal Path ⟶ Boolean
var: Xp : Path;
valid?(Self, Xp) == not(null?(Xp)) andthen name(Self) = head(Xp)
                                   and null?(tail(Xp))
-------------------------------------------------------------
;; environment : compute the lexical environment
   environment : Terminal Path ⟶ DecList
var: Xp : Path;
requires: valid?(Self)
environment(Self, Xp) == declarations(Self)
-------------------------------------------------------------
...
```

Figure 7: Some `Terminal` Extensions

The simplest way is to use a **functional presentation** like in methods of figure 7. The general form for the m method is: `Condition ==> m(Self, *) == Term`. Other presentations are not frequent in functional programming but common to algebraic specifications.

There is a primitive generator: `new<CFC>`, but it is possible to define new generators using a **safe presentation**:

```
extensions
-------------------------------------------------------------
;; block : its result is the designated inner block
   block : Terminal Path ⟶ Terminal
requires: valid?(Self, Xp)
var: Xd : Path;
        name(block(Self, Xd)) == name(Self)
declarations(block(Self, Xd)) == declarations(Self)
```

Figure 8: A Safe Presentation

A safe presentation is a set of axioms, one for each field selector `fsel`: `Condition ==> fsel(m(Self, *)) == Term`.

A **G-derived** presentation is a set of axioms, one for each class generator `gen`: `Condition ==> m(gen(*)) == Term`. An example is the following definition for `all`:

```
                              extensions
;; all :  number of all the declarations
   all :  Terminal  --->  Integer
var:  Xd, Yd :  Declaration; Xs :  String;
all(newTerminal(name= Xs, declarations= Yd)) == length(Yd)
all(block(Self, Xd)) == length(declarations(Self))
```

Figure 9: A G-Derived Presentation

Note that in the last case the generators must be defined before any G-derived presentation.

Some methods have preconditions (`requires:`), for example `environment` in the `Node` class of figure 10. The programmer must ensure that the precondition is true before using the method. The idea is the same as in CLU, Modula-2 or Eiffel. Preconditions may also exist in field selectors as in figure 6.

```
                              Node
                     inherits from DecTable
comments:  class for blocks with inner blocks
features:  number, all, valid?, environment, ...
from:  FullDecTable import:  mapall, mapvalid?, mapenvironment
from:  DecList import:  concatandmask
from:  Path import:  null?
from:  ...  import:  ...
                      aspect :  complex block
```

field selectors	constraint
name : Node ⟶ String	
declarations : Node ⟶ DecList	
inners : Node ⟶ FullDecTable	

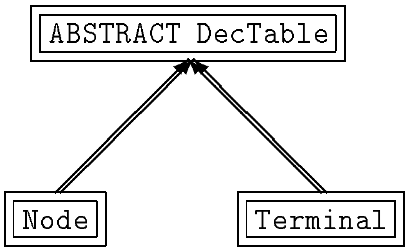
```
                              extensions
;; all :  number of all the declarations
   all :  Node  --->  Integer
all(Self) == number(Self) + mapall(inners(Self))

;; valid?  :  tests if the path is valid in the program
   valid?  :  Node Path  --->  Boolean
var:  Xp :  Path;
valid?(Self, Xp) == not(null?(Xp)) andthen name(Self) = head(Xp)
                                  and mapvalid?(inners(Self), tail(Xp))

;; environment :  compute the lexical environment
   environment :  Node Path  --->  DecList
var:  Xp :  Path
requires:  valid?(Self, Xp)
null?(tail(Xp)) == true ==> environment(Self, Xp) == declarations(Self)
null?(tail(Xp)) == false ==> environment(Self, Xp) ==
   concatandmask(declarations(Self), mapenvironment(inners(Self), tail(Xp)))

...
```

Figure 10: The `Node` Formal Class

Note that axioms allow the use of conditions as in the `environment` method. In many cases the axioms are simple, and the conditions are exclusive and complementary. This technique is widely used in functional programming or in algebraic specifications to simplify checking and proofs.

4 Inheritance

The inheritance scheme, restricted to aspects, is shown in figure 11.



ABSTRACT DecTable	
aspect : general block	
field selectors	**constraint**
name : ABSTRACT DecTable ⟶ String	
declarations : ABSTRACT DecTable ⟶ DecList	

Node	
aspect : complex block	
field selectors	**constraint**
name : Node ⟶ String	
declarations : Node ⟶ DecList	
inners : Node ⟶ FullDecTable	

Terminal	
aspect : simple block	
field selectors	**constraint**
name : Terminal ⟶ String	
declarations : Terminal ⟶ DecList	

Figure 11: Inheritance Scheme of `DecTable`

There is no inheritance of aspects as in programming languages. Aspects are used to check the inheritance rules (definition 4.2). In our model we use inheritance more rigorously than in concrete languages. This avoids some problems (inheritance of instance variables, confusing aggregation with inheritance, type errors and complexity of method lookup) and especially allows to define a minimal model for inheritance ([Roy92]).

4.1 Definition

Our idea of inheritance is based on a notion of cœrcion named structural projection.

Definition 4.1 (Structural Projection)
There is a structural projection from class C to class S (noted spC-S) iff for all field selector fsel of S and for all C instance c it exists a S instance s such that

- *the C constraint implies the S one,*
- *the C precondition of fsel implies the S one and*
- *either* `fsel(c) == fsel(s)` *or it exists a structural projection* `spR-T` *and* `spR-T(fsel(c)) == fsel(spC-S(s))`.

This defines a partial ordering and formalizes the well-known idea: an instance of the subclass may be view as an instance of the superclass. This property is often named the substituality principle or the subsumption rule or the conformance rule, it explicits instance polymorphism.

Definition 4.2 (Static Inheritance Criterion)
An inheritance link is possible between two classes if the following criterion is true: every field selector of the superclass exists in the subclass with the same type or a subtype.

We check that this criterion is true between `Terminal` and `DecTable` and between `Node` and `DecTable`. Then classes `Terminal` and `Node` can inherit from abstract class `DecTable`.

The previous criterion is necessary but not sufficient. If they are constraints or field selector preconditions, the rule implies additional constraints and additional conditions in the subclass. But in these cases no general and static criterion exists.

4.2 Using Inheritance

Using inheritance is as following:

1. first we must define the aspect of the class,
2. then we look for structural projection using the previous criterion:
   - if the result is true then an inheritance link can be defined.
   - if the result is false this can appear from two main cases:
     - the idea of inheritance is wrong.
     - the criterion is true after a renaming of the field selectors.

In any case recall that inheritance is correct if and only if methods of the superclass can be applied, without error, to instance of the subclass.

To avoid difficulties with method lookup we choose a general, simple and natural software engineering rule. Whenever there is a name clash, the designer must rename or, better redefine the method. Other strategies like depth-first, breadth-first or topological search are possible but they can add implementation problems. By now, we assume a complete dictionary of methods associated to each class and noted `K(C)`. This dictionary is the **complete behaviour** of the class coping with the lookup strategy.

Once an inheritance link is defined it implies the following principles:

- There is no exception, a method defined in a superclass is also visible in all its subclasses. Furthermore a public method must be public in every subclasses.
- A non abstract class has only non abstract methods in its complete behaviour.
- Only extensions are inherited, there is no implicit inheritance of primitive methods (field selectors, `new<CFC>`, etc.) or constraint.
- Redefining and masquing an extension is possible according to some covariant rule (see section 4.3).
- Inheritance implies subtyping.

4.3 Inheritance and Typing

There are two kind of interactions between typing and inheritance: typing problem arising from the type hierarchy [JKKM89] and subtyping.

The first one needs a rule to cure them:

Definition 4.3 (Unique Retraction Condition)
For all S and for all `m` *not in* `K(S)` *but visible for a subclass of* `S` *it exists a single subclass* `C` *of* `S` *such that for all subclasses* `R` *of* `S` *if* `m` *is in* `K(R)` *then* `R` *ako* `C`*.*

This condition is generally true but sometimes a graph modification is needed to ensure it.

Each class defines a new type and an inheritance link defines a corresponding (in the opposite way) subtype relation. To get a safe and static type checking we need additional rules for redefining methods. We do not handle functional objects (method as instance) this avoid the need of a contra-variant (or anti-monotonic) rule [Car84] which is not natural in OOP.

The simplest redefinition rule allows only a monotonic redefinition on the receiver type and the resulting type.

Definition 4.4 (Simple Covariance)
$m \ : \ S_1 \ S_2 \ ... \ S_n \longrightarrow S' \ \in K(S_1) \Longrightarrow \ \forall C_1 \ \leq \ S_1, \ \exists$ *a unique method*
$m \ : \ C_1 \ S_2 \ ... \ S_n \longrightarrow C' \ \in K(C) \ \wedge \ C' \leq S'.$

A natural extension of this rule is:

Definition 4.5 (Multiple Covariance)
$m \ : \ S_1 \ S_2 \ ... \ S_n \longrightarrow S' \ \in K(S) \Longrightarrow \ \forall C_i \ \leq \ S_i, \ \exists$ *a unique method*
$m \ : \ C_1 \ S_2 \ ... \ S_n \longrightarrow C' \ \in K(C) \ \wedge \ \forall 1 \leq j \leq n \ C_j \leq S_j \ \wedge \ C' \leq S'.$

Note that these rules are natural in OOP and consistent with our inheritance criterion which allows a monotonic redefinition of field selectors.

5 Type Checking

The model fits well to dynamically typed languages but also to strongly typed languages like Eiffel.

A first problem concerns terms like `head(tail(inners(newNode(...))))` which are meaningful but may be type erroneous depending on the type hierarchy. Our solution to this problem is similar to [JKKM89] and described in [AR94]. It needs the rule 4.3 and a first parsing of the program before the traditional type checking.

Our type checking is based on explicit declarations of type (variables and methods) and simple principles:

- There are, may be, predefined types (which are not classes!).
- Variables and methods are explicitly typed by the designer.
- A class defines a type and inheritance implies subtyping.
- Typing a message is like typing an operation application.

We add some redefinition rules (definitions 4.4 and 4.5) specific to OOP. An algorithm parses the program and checks the consistency of the type of expressions.

5.1 Type Expressions

Types are simply predefined types, classes and method types (profiles).

$$\texttt{Type ::= Basic} \mid \texttt{Class} \mid \texttt{Type*} \ \longrightarrow \texttt{Type}$$

5.2 Typing a Term

Typing a term requires a context (Γ) which contains the type of variables, method profiles and the subtype relations. The rules are usual for variables and operation applications. To compute the type of a message we use the following rule:

Definition 5.1 (Typing a Message)

$$\frac{\begin{array}{c}\forall i \ \Gamma \vdash e_i : C_i \leq S_i \\ \Gamma \vdash profile(m,\ C_1) = S_1\ S_2\ ...\ S_n \longrightarrow S\end{array}}{\Gamma \vdash m(e_1,\ ...,\ e_n) : S} \tag{1}$$

The mapping $profile(m,\ C_1)$ finds the method profile inherited by C_1 (in $\texttt{K(C}_1\texttt{)}$). For example: let a context

{`maindec : Declaration; blocks : FullDecTable;`
`newNode : String Declaration FullDecTable` $\longrightarrow$ `Node,`
`number : ABSTRACT DecTable` $\longrightarrow$ `Integer; Node < ABSTRACT DecTable`}, typing the message:

`number(newNode(name= ‘‘nom’’, declarations= maindec, inners= blocks))` needs typing `newNode(name= ‘‘nom’’, declarations= maindec, inners= blocks)` which is of type `Node`. The type of `number` is `ABSTRACT DecTable` $\longrightarrow$ `Integer` and `Node < ABSTRACT DecTable` then from the previous rule we deduce the type `Integer` for the entire expression.

5.3 Algorithm

The checking algorithm is really natural, an overview is:

Algorithm 5.2 (Type Checking)

`check : Program Context` $\longrightarrow$ `Boolean X Context`

`Program`	: *check classes and the main term in an empty context*
`Class`	: *enrich the context with inheritance relations and check constraint, preconditions and extensions*
`Extension`	: *enrich the context with variable declarations and profile then check precondition and axioms*
`Axiom`	: *check several equations*
`Condition`	: *check several equations*
`Equation`	: *check right and left term, they must have the same type*
`Term`	: *uses the rule for typing a term as previously explained*

More complex and user-friendly algorithms, such as type inference, are possible.

5.4 Safe Type Checking

We have proven some simple but useful results about this checking. If there is only simple covariance then the previous type checking is safe. More precisely if an expression is built according to well-typed classes and if its evaluation terminates then there is no dynamic type error during this evaluation. A dynamic type error is a message with an unknow selector (or a call with an unknow operation).

However multiple covariance is needed for method like `copy` or `add` in lists. But it can produce some typing problems with simple selection. These problems may arise in our functional model with `copy` but also in side effect languages as in Eiffel [Coo89]. There exist syntactic rules to cure them, however there are very strict.

It is possible to use multi-selection (multiple dispatch) rather than simple selection and then the previous (naturally extended) type checking is safe with multiple covariance. But multi-selection may add some other difficulties. A solution is to transform multiple covariant method into simple one using argument transposition and type relaxation. This is a good solution if it does not constraint users, for example the system can use this trick for some primitive methods (for example `equal?`).

A better way is to systematically redefine a method which directly uses a multiple covariant method. Other less strict rules and association types are under study.

6 Symbolic Evaluation and Implementation

We define a symbolic evaluator which is useful to test specification or to prove properties [ACCR94]. Furthermore we study translation to concrete object-oriented languages.

6.1 Evaluator

We give an abstract operational semantic to this model using conditional term rewriting. It needs orientation of axioms into conditional rewrite rules. The semantic of primitive methods is also defined through rewrite rules. This is simple for field selectors ($\texttt{fsel}_i\texttt{(new<CFC>(X}_i\texttt{, ..., X}_n\texttt{)) --> X}_i$), a little more complex for `copy`, `equal?` or `describe`.

A method is then interpreted as a set of rules. A class is a set of method rules hence the original program is view as a (inheritance) graph of set of set of rules.

A term in normal form is either a term build only on predefined types or a `new<CFC>` (more generally a generator) on normal form terms. The dynamic type (its class) of a term is given by the type of its normal form. The basic evaluation strategy is by value[4] as in OOP languages. More precisely this strategy first evaluates the receiver term and selects in the graph the method rules to apply. Let the example of figure 2:

```
head(inners(newNode(name= "main",
                    declarations= maindec,
                    inners= newFullDecTable(head= newTerminal(name= "foo",
                                                              declarations= foodec),
                                            tail= ...))))
```

where `foodec` and `maindec` are `DecList` terms in normal form.
We first evaluates `head(inners(newNode(*)))`, it implies evaluating `inners(newNode(*))`. The receiver term is in normal form and its dynamic type is `Node`. `inners(newNode(*))` is rewritten with the rule: `inners(newNode(name= X, declarations= Y, inners= Z)) --> Z` and leads to `newFullDecTable(*)`. Then the rule for `head` in `FullDecTable` is applied and give `newTerminal(name= ''foo'', declarations = foodec)`. The rule for `number` in `DecTable` is applied and give: `length(foodec)`. Abstract methods are avoided because in a well-typed program a message does not use an abstract method during its evaluation. These methods are useful to type program or to write more readable code.

6.2 Eiffel Translation

Rapid prototyping is a relevant tool for specification validation. The transition from formal classes to concrete classes is quite natural and partially automatic. Formal classes are simple to implement in concrete languages like CLOS, Smalltalk, Eiffel or C++. This stage is a concrete operational semantic for formal classes. Such translation takes as input the formal description and produces the "skeleton" of the class (class interface, class implementation, class structure, primitive methods code and extension signature).

[4]However a call-by-name may be possible using closure as in functional languages. Smalltalk uses this kind of mechanism with blocks.

The designer then builds on the concrete methods according to the semantics of secondary methods. Here automatic translation of extensions is possible because we have rewrite rules.

We experiment 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 hold for abstract methods. The second task is to write an *EXPORT* clause, it contains all the primitive methods and the features. Then for each superclass we compute an *INHERIT* clause with a *DEFINE* and a *REDEFINE* clause. The (re)definitions are used to avoid name clashes, the computation is done by a comparison of the behaviour of that superclass and the class. After that the main task is to define the *FEATURE*s.

We define private Eiffel attributes associated to each field selectors which is new or specialized in the subclass. Each field selector becomes an Eiffel routine which reads this attribute. The field selector precondition becomes a *REQUIRE* clause of this routine.

We define a *create* procedure whose parameters are the class attributes (inherited, new or redefined). The `new<CFC>` is a functional call to *create*. The constraint may become a *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 extension we define an associated Eiffel routine whose profile is the extension one minus 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. Axioms 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` is *current* and a message like `m(Self, *)` is translated into `m(*)`. Strong typing is not a problem because our type checking is more strict than the Eiffel one.

Experiments showed that classes and methods are very small and simple. But due to the functional background, the program is definitely not efficient. A good implementation requires optimization techniques. Full description of implementation techniques are out of the scope of this paper.

7 Related Works and Conclusions

The previous model is strongly related to abstract data type [EM85, FJ92]. We use similar concepts and notations. This allows to adapt the notions of consistency and completion from algebraic specification. However the FC model is often more concrete than algebraic specifications. We do not separate inheritance and subtyping as in [Bre91]. The model is also close to abstract model languages like CLU, but here the representation is only an abstract structure. These are the price to pay if you want to define and to use inheritance. Other benefits are to get the principle of incremental extension for classes and to relax hierarchical constraints of abstract data types.

Our model is formal, but it is not based on existing formal model like Object-Z or Z++ (Z), VDM++ (VDM) or OOZE (OBJ3, Z). These languages can reuse proof environments but a semantics object features has to be found in term of the underlying language. To contrast to these models we do not have strong structures that will be refined. Some of these languages provide concurrency and dynamic behaviour mechanisms such as traces or temporal logic. Others have modular construct to group classes (module). VDM++ separate structural inheritance and behavioural inheritance, in our model structural inheritance is not de facto but must be verified. As in Z++ or VDM++, we do not handle object identities while OOZE or Object-Z provide them. These languages do not provide translation means to operational languages. Other languages like COLD allow functional and state based style.

This is a functional model but contrarily to the family of functional languages (Caml, ML, Miranda, Haskell, Hope, Scheme ...) we do not allow first class functions. Our model focus on classes, inheritance and method redefinition with static typing rules which are incompatible with typed higher-order functions. However adding some higher-order methods as iteration or composition seems useful and of great (but future) interest. A comparison with Haskell [HJW+92] is interesting because this is a typed functional languages with OOP concepts. In Haskell we find the notion of type class, instance, inheritance, method and overloading. But as quoted in [HF94] it is a different perspective. In fact traditional OO programmers will be confound by such a practice. "In constrast to OOP, it should be clear that types are not objects, and in particular there is no notion of an object's or type's internal mutable state. An advantage over OOP is that methods in Haskell are completely type-safe: any attempt to apply a method to a value whose type is not in the required class will be detected at compile time instead of at runtime. In other words, methods are not "look-up" at runtime but are simply passed as higher-order functions". We claim that our model has a safe and static type system with true dynamic binding for (one-order) methods.

The model is as powerful as the Eiffel language except on two features: side effect and association types (*like*). Futhermore this model is not a concrete language; it is more abstract and more strict (and safe) than Eiffel. The FC model has been extended to side effects. The side effect model adds sequencing (;) and the primitive method `modify!`. The side effects of a method are restricted to the receiver (clause *modifies* of other languages). This extension is useful to optimize a first functional design. Genericity can be simulated by inheritance as in [Mey88, AR94]. Multi-selection [BDG+88] is also possible.

FCs are modules in the sense of Modula-2 or CLU where the interface is a formal specification and bodies are concrete classes in different languages. A comparison with Modula-3 is difficult because we have not a true language with a rich set of constructions. The object model of Modula-3 is a kind of typed-actor model [Aba94], then objects have a proper behaviour. In contrast to Modula-3 there is an inheritance criterion and we accept specializations of method profiles.

We defined a minimal and abstract model for OOP. This model is a formal specification language, it is close to algebraic abstract data types but with a more operational flavor. We defined rules for inheritance, safe type checking and an abstract semantic based on term rewriting. This model permits formal proofs in a functional style and simplifies the proof of the model implementation. It is easy to implement such classes in OOP languages, however optimizations are needed.

Some extensions are under study: metaclasses, methods as objects and association types. These extensions increase the difficulties of type checking.

References

[Aba94] Martìn Abadi. Baby Modula-3 and a Theory of Objects. *Journal of Functional Programming*, 2(4), 1994. DEC Report, February 2, 1993.

[ACCR94] Pascal André, Dan Chiorean, Corina Cîrstea, and Jean-Claude Royer. Object-Oriented Design With Formal Classes. Submitted to Contim'94, International Conference on Technical Informatics, April 1994.

[AR94] Pascal André and Jean-Claude Royer. La modélisation des listes en programmation par objets. In Pierre Cointe, Christian Quiennec, and Bernard Serpette, editors, *Journées Francophones des Langages Applicatifs (JFLA'94)*, number 11 in Collection Didactique, pages 259–285, Noirmoutier, 31 janvier - 1 février 1994. INRIA.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988.

[Bre91] Ruth Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Number 562 in Lecture Notes in Computer Science. Springer Verlag, June 1991. Thesis, Universität Passau, Germany.

[Car84] Luca Cardelli. A semantics of multiple inheritance. *LNCS 173 "semantics of data type"*, 173:131–144, June 1984.

[Coo89] William R. Cook. A Proposal for Making Eiffel Type-safe. *The computer Journal*, 32(4):305–311, 1989.

[DD90] D. Duke and Roger Duke. Towards a Semantics for Object-Z. In D. Bjorner, C.A.R. Hoare, and H. Langsmaack, editors, *Proceedings of VDM'90*, pages 244–261. Springer Verlag, 1990.

[DVK92] Eugene Dürr and Jan Van Katwijk. VDM++ : A Formal Specification Language for Object Oriented Design. In Georg Heeg, Boris Magnusson, and Bertrand Meyer, editors, *Proceedings of TOOLS 7*, Conference, pages 63–78. Prentice Hall, 1992.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specifications 1 : Equations and Initial Semantics*, volume 6 of *EATCS Monography on Theoretical Computer Science*. Springer-Verlag, 1985.

[FJ92] L.M.G. Feijs and F.B.M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Gau90] Marie-Claude Gaudel. *Algebraic Specifications*, chapter 22, page chapeter 22. in Software Engineer's Reference Book, March 1990.

[GL87] John Guttag and Barbara H. Lisbov. *Abstraction and Specification in Program Development*. The MIT Press, second edition, 1987.

[HF94] Paul Hudak and Joseph H Fasel. A Gentle Introduction to Haskell. Tutorial, Department of Computer Science, Glasgow University, 1994.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Arvind, Brian Boutel, Jon Fairbain, Joseph Fasel, Maria M Guzman, Kevin Hammond, John Hughes, Thomas Jonhnsson, Richard Kierburtz, Rishiyur S Nikhil, Will Partain, and John Peterson. Report on the Functional Programming LanguagesHaskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hor78] John Guttag J.J. Horning. The algebraic Specification of Abstract Data Types. *Acta informatica*, 10:27–52, 1978.

[JKKM89] Jean-Pierre Jouannaud, Claude Kirchner, Hélène Kirchner, and Aristide Megrelis. Programming with Equalities, subsorts, overloading and parameterization in OBJ. Technical Report 89-R-224, CRIN, Vandœuvre-les-Nancy, 1989.

[JL86] Jean-Pierre Jouannaud and Pierre Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, 1986.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* C.A.R Hoare Series. Prentice-Hall International, 1986.

[LH92] Kevin Lano and Howard Haughton. Reasoning and Refinement in Object-Oriented Specification Languages. In O. Lehrman Madsen, editor, *Proceedings of ECOOP '92*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer Verlag, 1992.

[Lin89] Charles Lins. *The Modula-2 Software Component Library*, volume 1. Springer Verlag, 1989.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Mus80] David Musser. On Proving Inductive Properties of Abstract Data Types. *in Proceedings of the Seventh Annual ACM Symposium on PoPL*, pages 154–162, January 1980. Las Vegas.

[Roy92] Jean-Claude Royer. A new set interpretation for the inheritance relation and its checking. *ACM OOPS MESSENGER*, 3(3):22–40, July 1992.