— DRAFT —

Overview of the C Object System *

Using C as an high-level object-oriented language

Laurent Deniau
CERN – European Organization for Nuclear Research
laurent.deniau@cern.ch

Abstract

The C Object System (COS) is a recent framework entirely written in C which implements high-level concepts available in CLOS, OBJECTIVE-C and other object-oriented programming languages: *uniform object model* (class, metaclass and property-metaclass), *generics, multimethods, delegation, exceptions, contracts* and *closures*. It relies on the programmable capabilities of C to extend its syntax and to implement the aforementioned concepts as *first-class objects*. COS aims at satisfying several general principles like *simplicity, flexibility, extensibility, efficiency* and *portability* which are rarely met in a single programming language. Its design is tuned to provide efficient and portable implementation of *message dispatch* and *message forwarding* which are the heart of code flexibility and extensibility. With COS features, software should become as flexible and extensive as with scripting languages and as efficient and portable as expected with C programming. Likewise, COS concepts should significantly simplify *adaptive*, *aspect-oriented* and *subject-oriented* programming as well as *distributed* systems.

Categories and Subject Descriptors D.3.3 [*C Programming Language*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming.

General Terms Object-oriented programming.

Keywords Adaptive object model, Aspects, Class cluster, Closure, Contract, Delegation, Design pattern, Dynamic class, Dynamic inheritance, Exception, Generic function, Introspection, High-order message, Message forwarding, Metaclass, Meta-object protocol, Multimethod, Multiple dispatch, Multiple inheritance, Predicate dispatch, Property-class, Uniform object model, Unit testing.

* COS project: `http://sourceforge.net/projects/cos`

1. Motivation

The C Object System (COS) is a small framework which adds an *object-oriented layer* to the C programming language [1, 2] using its *programmable capabilities*[1] while following the principles of *simplicity* of OBJECTIVE-C [4, 5, 6] and of *extensibility* of CLOS [7, 8, 9]. It is legitimate to ask what yet-another object-oriented language can bring to the community? Above all, COS aims to fulfill several general principles rarely met in a single programming language: *simplicity, flexibility, extensibility, efficiency* and *portability*. *Correctness* and *reliability* will be considered herein as obvious requirements that depend more on the quality of code, design and tests than on programming languages [10, 11].

1.1 Context

COS is developed in the hope to solve fundamental programming problems related to *applied metrology* [12, 13]. Although this domain looks simple at first glance, it involves nonetheless numerous fields of computer sciences; from *low-level* tasks like the development of drivers, protocols or state machines, the control of hardware, the acquisition of data, the synchronization of concurrent processes, or the numerical processing and modeling of dataset; to *high-level* tasks like the interactivity with databases or web servers, the management of remote or distributed resources, the visualization of dataset or the interpretation of scripts to make the system configurable and controllable by non-programmers [14, 15, 16]. Such systems have to process large dataset — up to few hundreds of megabytes per run — with high-bandwidth on machines sometimes limited. And in our case, only sparse human resources are available to develop *and* maintain such *continually-evolving-systems* (*i.e.* R&D). Therefore the challenge is ambitious and *if* our proposal succeeds to simplify the achievement of such systems, it could probably be useful to a wide variety of projects.

1.2 Principles

In a such context, it is essential to reduce the multiplicity of the technologies used, to simplify the development process, to enhance the productivity, the extensibility and the

[1] In the sense of "*Lisp is a programmable programming language*", [3].

portability of the code and to adapt the required skills to the available resources. Hence, the qualities of the programming language are preponderant in the success of such projects.

Simplicity The language should be easy to learn and use. The training time for an *average* programmer should be as short as possible what implies in particular a clear and concise syntax. Simplicity should become an asset which guarantees the quality of the code and allows to write complex constructions without being penalized by a complex formalism or by the multiplicity of the paradigms.

Flexibility The language should support code flexibility, namely the ability to reuse or quickly adapt existing code to unforeseen tasks. It is easier to achieve this goal if the language allows to write *generic code*, either by parameterization, either by abstraction of types.

Extensibility The extensibility or *long term flexibility*, is the *most demanding* criterion. The language must support the addition of new features or the improvement of existing features without changing significantly the code or the software architecture. Concepts like *dynamic typing*, *dynamic dispatch*, *dynamic object models* and *open object models* help to achieve good *extensibility* and *flexibility* by reducing couplings, but they are also the heel of Achilles of *efficiency*.

Efficiency A general purpose programming language must be efficient, that is it must be able to translate all kinds of algorithms into programs running with *predictable* resources usages (mainly CPU and memory) *consistent* with the processes carried out. In this respect, programming languages with an abstract machine close to the physical machine — a *low-level* language — offer generally better results.

Portability A general purpose programming language must be portable, that is it must be widely available on many architectures and it must accessible from almost any other languages (FFI). This point often neglected brings many advantages: it improves the software reliability, it reduces the deployment cost, it enlarges the field of potential users and it helps to find trained programmers. As regards this point, *normalized* programming languages (ISO) get the advantage.

1.3 Proposition

COS extends the C programming language with concepts [17] mostly borrowed from OBJECTIVE-C and CLOS. The absence of a specific compiler allowed to quickly explore various object models[2] while the Ralph E. Johnson's paper [18] was focusing the research towards the final design:

"If a system is continually changing, or if you want users to be able to extend it, then the Dynamic Object Model architecture is often useful. [...] Systems based on Dynamic Object Models can be much smaller than alternatives. [...] I am working on replacing a system with several millions lines of code with a system based on a dynamic object model that I predict will require about 20,000 lines of code. [...] This makes these systems easier to change by experts, and (in theory) should make them easier to understand and maintain. But a Dynamic Object Model is hard to build. [...] A system based on a Dynamic Object Model is an interpreter, and can be slow.".

This model [19, 20] seems to match exactly our needs and COS should provide the required features to *simplify significantly* the design of such systems *without efficiency loss*. In particular, COS has been designed to support efficiently two key concepts — *multimethods* and *fast generic delegation* — and provides a *uniform object model* where classes, metaclasses, generics and methods are *first-class objects*. Incidentally, COS strengthens inherently all the guidelines stated in [21] to build "*flexible, usable and reusable object-oriented frameworks*" as well as architectural pattern proposed in [22] to design *flexible component-based frameworks* .

2. COS in a Nutshell

COS is a small framework entirely written in portable C99[3] which provides programming paradigms like *objects, classes, metaclasses, generics, multimethods, delegation, exceptions, contracts* and *closures*. COS features are directly available at the source code level through the use of the keywords summarized in table 1 and defined in the header file `cos/Object.h`, and supported by its runtime library.

2.1 Concepts

Dynamic dispatch This concept available in *dynamic programming languages* is the heart of software extensibility because it postpones at runtime the resolution of methods invocation and reduces couplings between the callers and the callees. COS generalizes dynamic dispatch to *efficient multiple dispatch* and *fast message forwarding* (section 9).

Dynamic typing Dynamic dispatch requires dynamic typing to work properly what enhances *genericity* and reduces significantly code size and complexity. On one hand, these simplifications usually *improve the programmer understanding* who makes less conceptual errors, draws simpler designs and increases its productivity. On the other hand, dynamic typing postpones at runtime the detection of type errors with the risk to see programs ending prematurely. COS relies on the type system of C (section 3.4) to detect monomorphic type errors and provides *contracts* (section 5.3) and *messages tracer* to track dynamic types errors.

Encapsulation and separation Encapsulation is a major concern when developing libraries and large-scale projects. COS enforces encapsulation of both objects attributes and classes implementation because encapsulation is not only a matter of member access control but also a design issue. Besides, the object behaviors are represented by generics

[2] Within the limits of what is possible with the C preprocessor.

[3] Namely C89, variadic macros, [inline functions, compound literals]$_{opt}$.

Keywords[4]	Alternate keywords	Forms
useclass()[5]	COS_CLS_USE()	macro
defclass()[5 6]	COS_CLS_DEF()	macro
endclass[5]	COS_CLS_END	macro
makclass()[5]	COS_CLS_MAK()	macro
usegeneric()[5]	COS_GEN_USE()	macro
defgeneric()[6]	COS_GEN_DEF()	macro
defgenericv()	COS_GEN_DEFV()	macro
makgeneric()	COS_GEN_MAK()	macro
makgenericv()	COS_GEN_MAKV()	macro
usemethod()	COS_MTH_USE()	macro
defmethod()[6]	COS_MTH_DEF()	macro
endmethod	COS_MTH_END()	macro
retmethod()	COS_MTH_RET()	macro
next_method()[6]	COS_MTH_NXT()	macro
forward_message()[5]	COS_MTH_FWD()	macro
self[5]{1..4}	COS_MTH_SLF({1..4})	macro
retval[5]	COS_MTH_RETVAL	macro
TestType()	COS_CTR_TYP()	macro
TestAssert()	COS_CTR_ASS()	macro
TestInvariant()	COS_CTR_INV()	macro
PRE[7]	COS_CTR_PRE	macro
POST[7]	COS_CTR_POST	macro
BODY[7]	COS_CTR_BODY	macro
TRY[5]	COS_EX_TRY	macro
CATCH()[5]	COS_EX_CATCH()	macro
FINALLY()[5]	COS_EX_FINALLY()	macro
ENDTRY	COS_EX_ENTRY	macro
THROW()[5]	COS_EX_THROW()	macro
RETHROW()[5]	COS_EX_RETHROW()	macro
PRT()	COS_EX_PRT()	macro
UNPRT()	COS_EX_UNPRT()	macro
YES[5], NO[5], NIL[5]	COS_{YES,NO,NIL}	macro
True, False, Nil	---	useclass
S32, U32, S64, U64	---	typedef
BOOL[5], STR[5], FUNC	---	typedef
OBJ[5], CLASS[5], SEL[5]	---	typedef
IMP[5]{1..4}	---	typedef

Table 1. COS keywords and alternate keywords.

what favors the separation of interfaces and reduces cross-interfaces dependencies [21]. Table 2 lists different levels of encapsulation as types become more abstract and generic and of separation as invocations become more dynamic: *concrete*, *abstract* (ADT), *parametric* (C++ templates), *polymorphic* (C++ virtuals), *protocol* (JAVA interfaces), *dynamic* (CLOS & COS generics). An open model indicates that new functions or methods can be added separately.

Ownership The management of objects lifetime requires a clear policy of ownership and scope. In languages like C and C++ where semantic *by value* prevails, the burden is put on the programmer's shoulders. In languages like JAVA, C# and D where semantic *by reference* prevails, the burden is put on the garbage collector. In this domain, COS lets the developer to choose between garbage collection (Boehm GC [23]) and *manual reference counting with rich semantic* (section 3.5).

[4] A macro keyword is disabled if COS_DISABLE_*keyword* is defined.

[5 6 7] Borrowed or adapted respectively from OBJECTIVE-C, CLOS and D.

Type	Invocation	Dependency	Model
concrete	direct	interface & data	open
abstract	direct	interface	open
parametric	direct	implementation	open
polymorphic	indirect	interface	closed
protocol	indirect	interface (small)	closed
dynamic	lookup	generic (single)	open

Table 2. Type abstraction *vs.* interface extensibility.

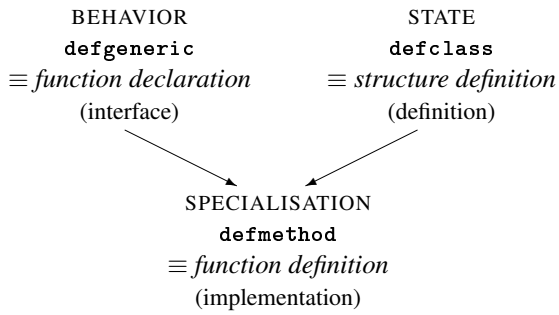

Figure 1. Roles of COS components and their equivalent C-forms. *Multi*methods are class*es* specialization of generics.

Concurrency COS has been designed from the beginning with concurrency in mind and shares only its dictionary of *static components*. Per thread resources like *messages caches* or *autorelease pools* rely on either thread-local-storage or thread-specific-key according to the availability.

2.2 Components

The object-oriented layer of COS is based on three components (figure 1) borrowed from CLOS which characterize the *open object model* well described in [9].

Classes Classes play the same role as *structures* in C and define objects *attributes*. They are bound to their *superclass* and *metaclasses* and define supertypes-subtypes hierarchies.

Generics Generics play the same role as *functions declarations* in C and define *messages*. They are essential actors of code extensibility and ensure correctness of messages formal parameters between the callers and the callees.

Methods Methods play the same role as *functions definitions* in C and define *generics specializations*. A method is invoked if the message belongs to its generic and the arguments match its classes specialization (multimethods).

The similarities between COS components and their equivalent C-form let expect an immediate productivity of C programmers with basic notions of object-oriented design. The

open object model allows to define components in different places and therefore requires an extra linking iteration to collect their *external symbols*: link → collect[8] → re-link. This fast iteration is automatically performed by the makefiles coming with COS before the final compilation stage that builds the program or the dynamic library.

3. Classes

COS allows to define and use classes as easily as in other object-oriented programming languages using the user-friendly syntax summarized in figure 2.

3.1 Using classes

In order to have direct access to classes as *first-class objects*, one can use the `useclass()` *declaration*. To highlight the similarities between OBJECTIVE-C and COS, let's start with a simple program:

```
1  #include <cos/Object.h>
2  #include <cos/generics.h>
3
4  useclass(Counter, (StdoutStream)Out);
5
6  int main(void) {
7    OBJ cnt = gNew(Counter);
8    gPut(Out,cnt);
9    gRelease(cnt);
10 }
```

which can be translated line-by-line into OBJECTIVE-C by:

```
1  #include <objc/Object.h>
2  // Counter interface isn't exposed intentionally
3
4  @class Counter, StdoutStream;
5
6  int main(void) {
7    id cnt = [Counter new];
8    [StdoutStream put: cnt];
9    [cnt release];
10 }
```

Line 2 makes the standard generics like `gNew`, `gPut` and `gRelease`[9] visible in the current translation unit. OBJECTIVE-C doesn't need this information since methods are bound to their class but if the users want to be warned for incorrect uses of messages, the class definition must be visible. This example shows that COS requires less information than OBJECTIVE-C to handle compile-time checks what leads to better code insulation and reduces useless recompilations. Moreover, it offers tighter tuning of interfaces exposure since only used messages need to be declared. The line 4 declares the class `Counter`[10] and the alias `Out` in replacement of the class `StdoutStream`, both classes being supposedly defined somewhere else otherwise a link-time error would occur. In line 7, the generic type `OBJ` is equivalent to `id` in OBJECTIVE-C, or `var`, `let` or `my` in other dynamic languages. The lines 7 – 9 show the life cycle of objects, starting with `gNew` (resp. `new`) and ending with `gRelease` (resp. `release`). They also show that generics *are* functions (*e.g.* one can take their address), a positive point to speedup the training of C programmers. Finally, the line 8 shows an example of multimethod where the message `gPut(_,_)` will look for the specialization `gPut(pStdoutStream,Counter)` whose meaning is discussed in section 5.

[8] COS mangled symbols are collected with the `nm` command or equivalent.

[9] By convention, the name of generics always starts by a '`g`' or a '`v`'.

[10] By convention, the name of classes always starts by an uppercase letter.

class-declaration:
  useclass(*class-decl-list* **);**

class-decl-list:
  class-decl
  class-decl-list **,** *class-decl*

class-decl:
  class-name
  (*class-name* **)** *alternate-name*

class-definition:
  defclass(*class-specifier* **)**
↪ *struct-declaration-list*[11]
↪ **endclass**

class-instantiation:
  makclass(*class-specifier* **);**

class-specifier:
  class-name
  class-name **,** *superclass-name*
  rootclass-name **,** _

{*class, superclass, rootclass, alternate*}*-name:*
  identifier[12]

Figure 2. Syntax summary of classes.

3.2 Defining classes

The definition of a class is very similar to a C structure:

```
defclass(Counter)
  unsigned val;
endclass
```

which is translated in OBJECTIVE-C as:

```
@interface Counter : Object {
  unsigned val;
}
// declaration of Counter methods not shown
@end
```

[11] [12] Defined respectively in §6.7.2.1 and 6.4.2.1 of [1].

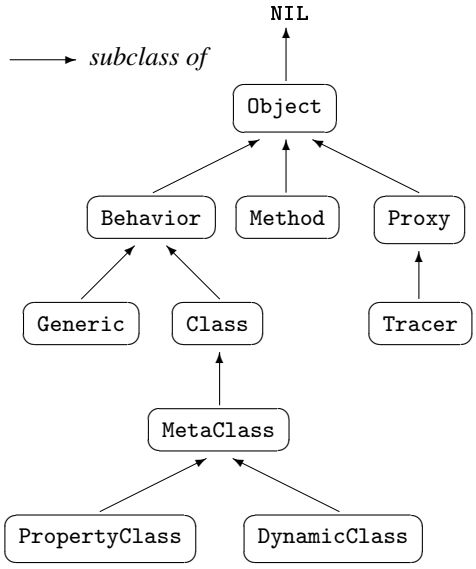

Figure 3. Subset of COS core classes hierarchy.

or equivalently in CLOS as:

```
(defclass Counter ()
  ((val)) )
```

The `Counter` class derives from the root class `Object` — the default behavior when the superclass isn't specified — and defines the `val` attribute for all its instances[13].

Class visibility *What must be visible and when*? COS allows three levels of visibility: none, declaration and definition. If you only use the generic type `OBJ`, nothing is required:

```
static inline OBJ gNew(OBJ cls) {
  return gInit(gAlloc(cls));
}
```

If you want to create instances from class declaration, only the declaration is required:

```
static inline OBJ gNewBook(void) {
  useclass(Book); // local declaration
  return gNew(Book);
}
```

If you want to access to objects attributes or define new subclasses, new methods or objects with automatic storage duration, the class definition must be visible.

3.3 Class inheritance

Class inheritance is as easy in COS as in other object-oriented programming languages. Figure 3 shows the hierarchy of the core classes of COS deriving from the root class `Object`. As an example, the `MilliCounter` class defined hereafter derives from the class `Counter` to extend its resolution to thousandths of count:

```
defclass(MilliCounter,Counter)
  unsigned mval;
endclass
```

which gives in OBJECTIVE-C:

```
@interface MilliCounter : Counter {
  unsigned mval;
}
// declaration of MilliCounter methods not shown
@end
```

and in CLOS:

```
(defclass MilliCounter (Counter)
   ((mval)) )
```

In the three cases, the derived class inherits of the attributes and the methods of its superclass. This implies that the definition of the superclass must be visible what creates a strong coupling between the two classes. Since COS aims at insulating classes as much as possible, it discourages direct access to superclass attributes by introducing a syntactic indirection which forces the user to write *obj*`->Super.attribute` instead of *obj*`->attribute`. The inheritance of *multimethods* has a different meaning and will be discussed in section 5.

Root class Defining a root class is an exceptional task and requires some precautions in dynamic languages, but it may be a necessity in some rare cases. COS uses the terminal symbol $\perp$ (represented by '`_`') to mean "*end of hierarchy*" and to declare a class as a root class[14]. The hierarchies shown in figures 3 and 6 have two important root classes with rather simple definitions:

```
defclass(Object,_) endclass
defclass(Nil,_) endclass
```

On the other hand, the definitions of their methods must be written with care since they must provide essentials functionalities inherited by their subclasses.

Class rank COS computes at compile-time the inheritance depth of each class, namely the number of its superclass. The rank of a root class is one (by definition) and each successive subclassing increases the rank by one. The method `gRank(`*cls*`)` returns the rank of *cls*. The inheritance depth is limited to rank 64 which should be far enough in practice.

Dynamic inheritance COS provides the message `gChangeClass(`*obj*`,`*cls*`)` to change *effectively* the class of *obj* to *cls* *if* it is a superclass of *obj*'s class *and* the instances sizes of both classes are equal; and the message `gUnsafeChangeClass(`*obj*`,`*cls*`)` to change *effectively* the class of *obj* to *cls* *if* both classes share a common non-root superclass *and* the instance size of *cls* is lesser or equal to the size of *obj*.

[13] Objects are aggregation (*i.e.* C `struct`) in COS.

[14] Technically a root class derives from `NIL`.

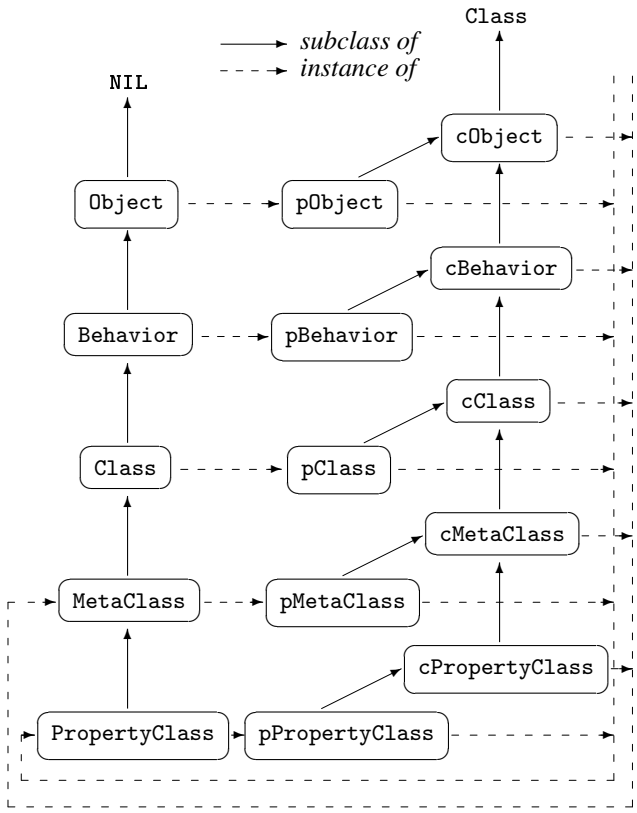

Figure 4. COS core classes hierarchy with metaclasses.

3.4 Meta classes

In COS (as in OBJECTIVE-C), classes definition create a parallel hierarchy of metaclasses which standardizes the use of classes as first-class objects. Figure 4 shows the complete hierarchy of the `PropertyClass` class.

Class metaclass The metaclasses are *classes of classes* implicitly defined in COS to ensure the coherency of the type system: to each class must correspond a metaclass [24]. Both inheritance trees are built in parallel: if a class `D` derives from a class `B`, then its metaclass `cD`[15] derives from the metaclass `cB` — excepting root classes which derive from `NIL` and have their metaclasses deriving from `Class`. Metaclasses are instance of the `MetaClass` class.

Property metaclass In some cases (*e.g.* classes clusters), automatic derivation of the class metaclass from its superclass metaclass can be problematic. To solve the problem COS creates for each class, a *property metaclass which cannot be derived*; that is all methods specialized on the property metaclass can only be reached by the class itself. In order to preserve the consistency of the hierarchy, a property metaclass must always derive from its class metaclass [25],

[15] The metaclass name is always the class name prefixed by a '`c`'.

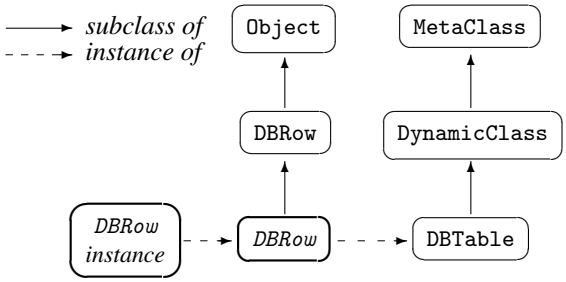

Figure 5. Example of DOM to model results of DB queries.

namely `pB`[16] (resp. `pD`) derives from `cB` (resp. `cD`). Property metaclasses are instance of the `PropertyClass` class.

Dynamic metaclass In section 1, we have been sensitized to the importance of the dynamic object model (DOM) and its underlying dynamic behavior that classes statically defined cannot achieve easily [20, 26]. For this purpose, COS provides the metaclass `DynamicClass` (figure 3) which can be derived by user-defined metaclasses to build specialized classes *at runtime*. The strategy of such metaclasses [24] is to deal dynamically with instances that differ slightly on their attributes and share behaviors inherited from a common superclass [18]. Figure 5 shows an example of a simple DOM application where *results of database queries* are represented by dynamic classes. The `DBTable` metaclass creates *on the fly* an anonymous subclass *`DBRow`* of the `DBRow` class which *represents the row structure of the result*, namely the columns properties. Each new query will create a new subclass of `DBRow` since the columns properties are only known at runtime. The anonymous subclasses *`DBRows`* will inherit their methods from `DBRow` to create and manipulate per-row instances. Thanks to reference counting, the anonymous classes will be automatically destroyed when their last instances (rows) will be destroyed.

Class predicate With multimethods, it is possible to generalize the usage of metaclasses and define classes specifically for this purpose. Figure 6 shows the hierarchy of the core class-predicates used in COS to specialized multimethods to specific *states*. For instance messages like `gAnd`, `gOr` and `gNot` are able to deal with expressions containing the class-predicates `True`, `False` and `TrueFalse`. The root class `Nil`[17] is a special case of *absorbent element* which silently ignores all received messages — *an inherited behavior*.

Type system From the point of view of static typing, COS follows the same rules as OBJECTIVE-C except that multimethods reduce significantly the need for runtime identification of generic type. From the point of view of dynamic typing, the set of *class – metaclass – property-class* forms a co-

[16] The property metaclass name is always the class name prefixed by a '`p`'.

[17] `Nil` and `NIL` aren't the same since sending a message to `Nil` is safe while sending a message to `NIL` crashes the program.

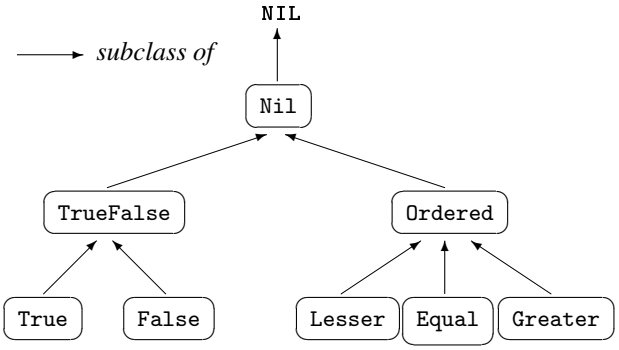

Figure 6. Subset of COS core class-predicates hierarchy.

herent hierarchy of classes and types which offers more *flexibility* than those of OBJECTIVE-C and SMALLTALK. The set of *class – metaclass – dynamic-class* forms a coherent hierarchy of classes which offers more *extensibility* towards adaptive object models.

3.5 Class instances

Objects lifespan The life cycle of objects in COS is very similar to other object oriented programming languages, namely it *starts* by allocation (`gAlloc`) followed by initialization (`gInit` and variants) and *ends* with *de*initialization (`gDeinit`) followed by deallocation (`gDealloc`). In between, the users manage the ownership of objects — their *dynamic scope* — with `gRetain`, `gRelease` and `gAutoRelease` like in OBJECTIVE-C. In principle, user-defined classes inherit `gAlloc` and `gDealloc` from `Object` and need only to define at least a *constructor* — that is messages with name starting by `gInit` — and the *destructor* `gDeinit`. The *copy constructor* is the specialization of the message `gInitWith(_,_)` for the same class twice.

Objects type In COS (resp. OBJECTIVE-C), objects are always of dynamic type because the type of `gAlloc` and `gInit` (resp. `alloc` and `init`) is `OBJ` (resp. `id`). Since it is the first steps of the objects life cycle in both languages, the type of objects can never be known statically. That is why COS (resp. OBJECTIVE-C) provides the message `gIsKindOf(`*obj*`,`*cls*`)` (resp. `[`*obj* `isKindOf:` *cls*`]`) which returns `True` (resp. `YES`) if the object *obj* is an instance of a subclass of *cls*, `False` (resp. `NO`) otherwise. But even so, it would be dangerous to use static cast to convert the object to the expected type because of dynamic design patterns like class clusters and proxies. To this effect, COS provides the message `gCastTo(`*obj*`,`*cls*`)` which *ensures* that the *returned* object is of the expected type upon success (*i.e.* `struct` *`Class*`*) and `Nil` otherwise. The cast can have various effects amongst *downcast* (object type is checked through inheritance), *coercion* (object type is changed)[18], *conversion* (new object is created and attributes are converted)[19] and *substitution* (object is substituted by another one)[20].

[18] This may occurs if the object is member of a *class cluster*.

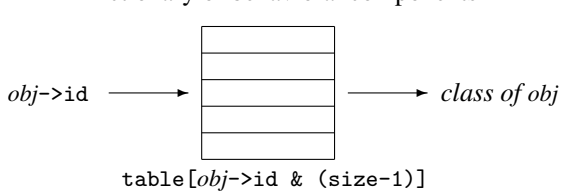

Figure 7. Lookup to retrieve object's class from object's id.

Objects identity COS provides the message `gClass(_)` which returns the class of an object; a metaclass if the object is a class. An object is bounded to its class through a unique 32 bits identifier. Figure 7 shows how this number is used to retrieve efficiently the class of an object from the dictionary of behavioral components. Comparing to implementations based on pointers to bound objects to classes, the unique identifier has three advantages: it ensures better behavior of lookup caches under heavy load, it improves message dispatch since no access to classes is required[21] and it is smaller than pointers on 64 bits architectures.

Automatic objects Since COS adds an object oriented layer on top of C, it is easy to create objects with automatic storage duration (*e.g.* on the stack) with compound literals[22]. In order to achieve this, the class definition must be visible and the developer of the class must provide a special constructor as a macro. For example the constructor `aStr(''a string'')`[23] is equivalent to the OBJECTIVE-C directive `@''a string''`. COS already provides automatic constructors for many small objects like `Bool`, `Char`, `Short`, `Int`, `Long`, `Double`, `Complex`, `Size`, `Index`, `Slice`, `Range`, `Point`, `Pointer`, `AllocPointer`, `Function`, `Functor`[24] and `Tuple`. These constructors allow to create efficiently temporary objects with local scope and enhance the genericity and the flexibility of the multimethods. For example, the constructor message `gInitWith(_,_)` and variants can be used in conjunction with almost all automatic constructors aforementioned. Thanks to the rich semantic of COS reference counting, if an automatic object receives the message `gRetain` or `gAutoRelease`, it is automatically cloned using the `gClone` message and the new copy with dynamic scope is returned.

Static objects Static objects can be built in the same way as automatic objects except that it requires one more step. It is worth to know that all COS *components* have static storage

[19] The new object should have been *autoreleased* before being returned.

[20] This may occur if the object is a *proxy* and the *delegate* is returned.

[21] Which for multimethods could require to access up to 4 classes.

[22] One can find interesting usages of *compound literals* in §7.4.5 [2].

[23] By convention, *automatic* constructors always starts by an 'a'.

[24] `Function` is a wrapper to function pointer while `Functor` is a closure.

duration and consequently are *insensitive to ownership* since their lifetime exceeds any dynamic scope.

3.6 Implementing classes

Classes instantiation create the *class* objects using the keyword `makclass` and the same *class-specifier* as the corresponding `defclass`. COS checks at compile-time if both definitions match. The counters implementation follows:

```
#include ''Counter.h''
#include ''MilliCounter.h''

makclass(Counter);
makclass(MilliCounter,Counter);
```

which is equivalent in OBJECTIVE-C to:

```
#include ''Counter.h''
#include ''MilliCounter.h''

@implementation Counter
// definition of Counter methods not shown
@end
@implementation MilliCounter
// definition of MilliCounter methods not shown
@end
```

Abstract class An abstract class is a class which doesn't define constructor and therefore cannot initialize instance.

Final class A final class is a class whose definition is part of its implementation and therefore cannot be made visible.

Class initialization For the purpose of pre-initialization, COS ensures to invoke *once by ascending class rank* (superclass first) all specializations of the message `gInitialize` on *property* metaclass before the first message is sent. Likewise, COS ensures to invoke *once by descending class rank* (subclasses first) all specializations of the message `gDeinitialize` on *property* metaclass after exiting `main`.

4. Generics

We have already seen in previous code samples that generics can be used as functions. But generics have in fact multiple forms and define each:

- a *function declaration* used to ensure at compile-time the correspondence of the definitions between a method (`defmethod`) and the generic it belongs to (`defgeneric`).
- a *function definition* used to dispatch the message what means looking for the most specialized method belonging to the generic which matches the classes of the objects used as *selectors*.
- a *message selector* of type `SEL` used by the dispatcher.

Figure 8 summarizes the syntax of generics which is half way between the syntax of generics definition in CLOS and the syntax of methods declaration in OBJECTIVE-C.

[25] Defined in §6.7.6 of [1]

generic-declaration:
`usegeneric(` *generic-decl-list* `);`

generic-decl-list:
generic-decl
generic-decl-list `,` *generic-decl*

generic-decl:
generic-name
`(` *generic-name* `)` *alternate-name*

generic-definition:
`defgeneric(` *generic-specifier* `);`

generic-instantiation:
`makgeneric(` *generic-specifier* `);`

generic-variadic-definition:
`defgenericv(` *generic-specifier* `, ...);`

generic-variadic-instantiation:
`makgenericv(` *generic-specifier* `, ...);`

generic-specifier:
return-type `,` *generic-name* `,` *selector-list*
return-type `,` *generic-name* `,` *selector-list* `,` *param-list*

selector-list:
discarded-name$_{opt}$
`(` *selector-decl-list* `)`

selector-decl-list:
discarded-name$_{opt}$
selector-decl-list `,` *discarded-name*$_{opt}$

param-list:
param-decl
param-list `,` *param-decl*

param-decl:
param-type
`(` *param-type* `)` *param-name*$_{opt}$

{*return, param*}*-type:*
type-name[25]

{*generic, param, discarded*}*-name:*
identifier

Figure 8. Syntax summary of generics.

Generic rank The rank of a generic is the number of formal parameters in its *selector-list*. COS supports generics from rank 1 to 4 what should be enough in practice since it covers 100% of the multimethods present in the standard libraries of CECIL and DYLAN [27, 28].

4.1 Message dispatch

COS dispatch uses global caches (one per rank) implemented with hash tables to speedup methods lookup. The caches solve slots collisions by growing until they reach a user-defined upper bound of slots. After that, they use packed linked list incrementally built to a maximum length of 3 cells. Above this length, the caches start to forget cached methods — a required behavior when dynamic classes are supported. The lookup uses *fast asymmetric hash functions* to compute the cache slots and ensures uniform distribution even when all selectors have the same type or specializations on permutations exist.

Fast messages COS lookup is simple enough to allow some code inlining on the caller side to speedup message dispatch. Fast lookup is enabled *up to the rank* specified by `COS_FAST_MESSAGE` — *disabled* = 0 (default), *all* = 4 — before the generics definition (`defgeneric`).

4.2 Declaring generics

Generics declarations are less common than classes declarations but they can be useful when one wants to use generics as first-class objects. Since definitions of generics are more often visible than class definitions, it is common to alias their name as in the following short example:

```
void safe_print(OBJ obj) {
  usegeneric((gPrint)prn);
  if (gUnderstandMessage1(obj,(SEL)prn) == True)
    gPrint(obj);
}
```

which gives in OBJECTIVE-C:

```
void safe_print(id obj) {
  SEL prn = @selector(print);
  if ([obj respondsToSelector: prn] == YES)
    [obj print];
}
```

By convention, messages suffixed by `{1..4}` have one version per supported rank of generics.

4.3 Defining generics

Definitions of generics correspond to functions declaration in C and differ from OBJECTIVE-C methods declaration by the fact that they are not bound to classes (prefix '-') nor metaclasses (prefix '+'). The following definitions:

```
defgeneric(OBJ, gIncr, (_)); // rank 1, with ()
defgeneric(OBJ, gIncrBy, _, int); // rank 1, no ()
defgeneric(OBJ, gInitWith, (_,_)); // rank 2
defgeneric(OBJ, gGetAtIdx,(_),(size_t)idx);//rk 1
defgeneric(OBJ, gPutAt, (where,what,at)); //rnk 3
```

can be translated into CLOS as:

```
(defgeneric incr (obj))
(defgeneric incr-by (obj val))
(defgeneric init-with (obj src))
(defgeneric get-at-idx (container idx))
(defgeneric put-at (where what at))
```

All parameters can be optionally named including in the selector-list, but this is only informative and discarded.

Instantiation Generics instantiations should take place in implementation files and must match their definitions, except that `makgeneric` replaces `defgeneric` as for classes.

Variadic generics Variadic generics are handled by the variant `defgenericv` (resp. `makgenericv`) and require '...' (ellipsis) as their last formal parameter:

```
defgenericv(OBJ, vPrintFmt, (_,_), ...); // rank 2
defgenericv(OBJ, vPrintStr, (_), STR, ...);//rk 1
```

and equivalently defined in CLOS by:

```
(defgeneric print-fmt (stream fmt &rest args))
(defgeneric print-str (stream str &rest args))
```

5. Methods

Methods are defined using a similar syntax as generics and summarized in figure 9. The following code defines a method specializing the message `gIncr` for the class `Counter` whose definitions must be visible:

```
defmethod(OBJ, gIncr, (Counter))
  ++self->val;
  retmethod(_1);
endmethod
```

which in OBJECTIVE-C gives (within `@implementation`):

```
- (id) incr {
  ++self->val;
  return self;
}
```

`self` is intensionally mentioned to enhance clarity and similarities with COS where it *must be specified*.

Unnamed parameters Methods can have unnamed formal parameters that COS will automatically name *_n* where *n* is the parameter position, *i.e.* `_1`, `_2`, ... As a special case, the parameters of the *specializer-list* are always unnamed.

Methods specializers The arguments of the *specializer-list* can be equivalently accessed through `self`*n*[26] whose types correspond to their class specializer (*e.g.* `struct Counter*`) or through unnamed parameters *_n* whose types are `OBJ` with $1 \leq n \leq N \leq 4$ where N is the message rank. It is worth to understand that `self`*n* and *_n* are bound to the same object.

Methods return It is worth to note that returning from methods must be achieved using `retmethod()` instead of `return`, otherwise a compile-time error occurs.

[26] `self` and `self1` are equivalent.

method-declaration:
  usemethod(*method-decl* **);**

method-decl:
  method-name **,** *specializer-list*
  (*method-name* **,** *specializer-list* **)** *alternate-name*

method-definition:
  defmethod(*method-specifier* **)**
↪ *method-statement*
↪ **endmethod**

method-statement:
  statement[27]
  method-statement-with-contract

method-return-statement:
  retmethod(*expression*$^{28}_{opt}$ **);**

method-next-statement:
  next_method(*argument-expression-list*[29]**);**

method-forward-statement:
  forward_message(*argument-expression-list* **);**

method-specifier:
  return-type **,** *method-name* **,** *specializer-list*
  return-type **,** *method-name* **,** *specializer-list* **,** *param-list*

method-name:
  generic-name
  (*generic-name* **,** *alternate-generic-name* **)**

specializer-list:
  class-name
  (*specializer-decl-list* **)**

specializer-decl-list:
  class-name
  specializer-decl-list **,** *class-name*

specializer-param-name:
  self$\{$**1..4**$\}_{opt}$

anonymous-param-name:
  _$\{$**1..**$\}$

returned-value-name:
  retval

Figure 9. Syntax summary of methods.

Multimethods Multimethods are methods with a *selector-list* of more than one formal parameter. The following example defines the assign-sum operator which adds a `Double` to a `Complex`[30]:

```
defmethod(OBJ, gAddTo, (Complex,Double))
  self1->val += self2->val;
  retmethod(_1);
endmethod
```

Up to now, about half of COS generics have a rank ≥ 2 and cover more than 70% of the methods definition.

Variadic methods Methods specializing variadic generics must have a `va_list` as their last formal parameter:

```
defmethod(OBJ, vPrintStr, (FileStream),
↪       (STR)fmt, (va_list)ap)
  vfprintf(self->fp, fmt, ap);
  retmethod(_1);
endmethod
```

Class methods Class methods are methods specialized for classes deriving from `Class` what includes all metaclasses:

```
defmethod(void, gInitialize, (pMyClass))
  // do some initialization for MyClass.
endmethod

defmethod(OBJ, gCastTo, (Object,Class))
  retmethod(cos_object_cast(self1,self2));
endmethod
```

where `cos_object_cast(`*obj*`,`*cls*`)` is the counterpart of the C++ operator `dynamic_cast` and declared in the low-level API of COS.

Methods type Because of fast message forwarding (section 5.2), methods belonging to the same generics rank must have the same type internally (*i.e.* signature):

```
void (*IMP1)(void*,SEL,OBJ,...);
void (*IMP2)(void*,SEL,OBJ,OBJ,...);
void (*IMP3)(void*,SEL,OBJ,OBJ,OBJ,...);
void (*IMP4)(void*,SEL,OBJ,OBJ,OBJ,OBJ,...);
```

where the first parameter is a pointer to the returned value, the second parameter is the message selector used by the dispatcher, the `OBJ`s are the parameters of the *specializer-list* and the ellipsis holds the remaining *param-list* (if any). Because of the *default argument promotion*[31], *param-list* should not contain parameter compatible with `char`, `short` or `float`, otherwise a compile-time error should occur.

5.1 Next methods

The `next_method` principle borrowed from CLOS[32] is the answer to the problem of superclass(es) methods *call* (*i.e.*

[27] [28] [29] Defined respectively in §6.8, §6.5.17 and §6.5.2 of [1].

[30] `val` fields are respectively of types `double` and `double _Complex`.

[31] Defined in §6.5.2.2 of [1].

[32] Namely `call-next-method` and `next-method-p`.

late binding) in the presence of *multimethods*. The following sample code defines a specialization of the message `gIncrBy` for the class `MilliCounter` which adds thousandths of count to the counter:

```
1 defmethod(OBJ, (gIncrBy,gIncr), (MilliCounter),
2 ↪       (unsigned)mval)
3   self->mval += mval;
4   if (self->mval >= 1000) {
5     self->mval -= 1000;
6     next_method(self);
7   } else
8     retmethod(_1);
9 endmethod
```

which is equivalent to the OBJECTIVE-C code:

```
1 - (id) incrBy:(unsigned)mval
2 {
3   self->mval += mval;
4   if (self->mval >= 1000) {
5     self->mval -= 1000;
6     return [super incr];
7   } else
8     return self;
9 }
```

Line 6 shows how `next_method` replaces the message sent to `super`. By default, `next_method` calls the next method belonging to the same generic (*e.g.* `gIncrBy`) where *next* means the method with the higher *method rank* lesser than the method rank of the current method. In some cases, it is important to test for the existence of the *next method* before calling it: `if (next_method) next_method(..);`. It is worth to note that `next_method` transfers the returned value directly to the method caller. In the method, the returned value can still be accessed through `retval` after the call.

Alternate generics In the example above, the `Counter` class has no specialization for `gIncrBy`. That is why the line 1 specifies an *alternate generic*, namely `gIncr`, to which `next_method` should look at to search the next method. The alternate generic must have the *same rank* and *return type* as the generic, otherwise a compile-time error occurs.

Methods rank The method rank is a value computed at compile-time and used to build the list of methods specialization. Considering the four classes `A`, `B`, `C`, `D` with ranks a, b, c and d respectively. The rank of a method with `(A,B,C,D)` as its *specializer-list* is given by:

$$r_m = (((((a+b+c+d)\times 2^6+a)\times 2^6+b)\times 2^6+c)\times 2^6+d)$$

To compute other methods rank, it suffices to replace the rank of missing specializers by zero: $r_m(\texttt{A}) = (a \times 2^6 + a) \times 2^{6+6+6}$. Assuming for instance `A <: B <: C`, it is easy to compute the *method precedence list* for the set of all pairs of `A`, `B` and `C` by *descending rank*: `(A,A) (A,B) (B,A) (A,C) (B,B) (C,A) (B,C) (C,B) (C,C)`. COS methods rank has some nice properties: it provides natural *left-to-right precedence*, it is *non-ambiguous*, *monotonic* and *totally ordered* and it fits on a 32 bits word. The latter allows fast method search in case of cache miss but limits the class rank to 64.

5.2 Message forwarding

Message forwarding is a major feature of COS which was developed from the beginning with *fast generic delegation* in mind.

Unrecognized message Message dispatch performs runtime lookup to search message specializations. If no specialization is found, the message `gUnrecognizedMessage{1..4}` is sent with the same arguments as the *original sending, including the selector*. Hence these messages can be overridden to support message forwarding. The default behavior of `gUnrecognizedMessage{1..4}` is to throw the exception `ExBadMessage`.

Forwarding message Message forwarding has been borrowed from OBJECTIVE-C and extended to multimethods. The sample code below shows a very common usage of message forwarding:

```
1 defmethod(void,gUnrecognizedMessage1,(MyProxy))
2   if(gUndertstandMessage1(self->obj,_sel)==True)
3     forward_message(self->obj);
4   else
5     next_method(self);
6 endmethod
```

which can be translated line-by-line into OBJECTIVE-C by:

```
1 - (retval_t) forward:(SEL)sel :(arglist_t)args {
2   if ([self->obj respondsTo:sel] == YES)
3     return [self->obj performv:sel :args];
4   else
5     return [super forward:sel :args];
6 }
```

Here, `forward_message` and `next_method` work the same way, except that the former uses message dispatch and the latter uses late binding. Both propagate all the arguments, including the hidden parameters `_ret`, `_sel` and `_arg` which hold respectively a pointer to the returned value, the original selector and the `va_list` pointing to the original arguments of the *param-list*. *Fast forwarding* requires that methods with a *param-list* check whether the `va_list` has an indirection or not (*i.e.* a `va_list` pointing to a `va_list`) and extracts the arguments accordingly. As for `next_method`, `forward_message` transfers the returned value directly to the method caller and `retval` still allows to access it after the sending.

Special forwarding Since the *param-list* is managed by a `va_list`, it is possible to access to its arguments. In order to do this, COS provides introspective information on generics (*i.e.* signature) which allows to retrieve the arguments and the returned value. But this kind of need should be exceptional and this topic is beyond the scope of this paper.

Fast forwarding Since all methods belonging to generics with equal rank have the same C function signature, *it is safe to cache the message* `gUnrecognizedMessage{1..4}` in place of the unrecognized message. Hence, the next sending of the latter will result in a cache hit. This substitution achieves message forwarding at half speed of message dispatch, one dispatch for the unrecognized message and one dispatch for the forwarding. The class `Proxy` strongly relies on this efficiency with for example rank 2 forwarding defined as:

```
defmethod(void,
↪       gUnrecognizedMessage2, (Proxy,Object))
  forward_message(self1->obj,_2);
endmethod

defmethod(void,
↪       gUnrecognizedMessage2, (Object,Proxy))
  forward_message(_1,self2->obj);
endmethod
```

Absorbent element The class-predicate `Nil` is an *absorbent object*, in that it defines all its possible specializations of the message `gUnrecognizedMessage{1..4}` — namely $2^1 + 2^2 + 2^3 + 2^4 - 4 = 26$ methods — to *absorb and forget* (trace in debug mode) all the messages received.

Transparent element The class `Proxy` creates objects which aim to be as *transparent* as possible while behaving on behalf of their delegate — a behavior inherited by its subclasses. In order to achieve this behavior, the class `Proxy` overrides the message `gIsKindOf`, `gCastTo`, and all its specialization of `gUnderstandMessage{1..4}` and `gUnrecognizedMessage{1..4}`. The only way to know if an object is an instance of `Proxy` is (eventually) through the message `gClass`. User-defined proxies can be written from scratch or inherit from `Proxy` and override methods to achieve *less* transparent behavior.

5.3 Contracts

To quote Bertrand Meyer [29], the key concept of Design by Contract is "*viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations*". Most languages that support Design by Contract provide two types of statements to express the obligations of the caller and the callee: *preconditions* and *postconditions*. The caller must meet all preconditions of the message sent, and the callee (the method) must meet its own postconditions — the failure of either party leads to a bug in the software. In that way, *Design by Contract* (*i.e.* developer point of view) is the complementary tool of *Unit Testing* (*i.e.* user point of view) since both enhance the mutual confidence between developers and users. In practice, both are written by the same developers who become more sensitized to responsibilities of each party and write less intrusive and more objective Unit Tests — leading to better software reliability and design.

method-statement-with-contract:
  declaration-without-initializer[33]
↪ *pre-statement*$_{opt}$ *post-statement*$_{opt}$ *body-statement*

pre-statement:
  PRE *statement*

post-statement:
  POST *statement*

body-statement:
  BODY *statement*

test-assert-statement:
  TestAssert(*expression* **)**
  TestAssert(*expression* **,** *file* **,** *line* **)**

test-invariant-statement:
  TestInvariant(*object-expression*[34]**)**

test-type-statement:
  TestType(*object-expression, class-name* **)**
  TestType(*object-expression, class-name* **,** *file* **,** *line* **)**

Figure 10. Syntax summary of contracts.

To illustrate how contracts work in COS with the syntax summarized in figure 10, we can rewrite the method `gIncr`:

```
defmethod(OBJ, gIncr, (Counter))
  unsigned old_val;
  PRE old_val = self->val;
  POST TestAssert(self->val < old_val);
  BODY { /* same code as before */ }
endmethod
```

The `POST` statement `TestAssert` checks for counter overflow *after* the execution of the `BODY` statement and throws an `ExBadAssert` exception on failure. The variable `old_val` initialized in the `PRE` statement *before* the execution of the `BODY` statement, plays the same role as the `old` feature in EIFFEL. One can rewrite in the same way the method `gIncrBy`:

```
defmethod(OBJ, (gIncrBy,gIncr), (MilliCounter),
↪       (unsigned)mval)
  PRE TestAssert(mval <= 1000);
  POST TestInvariant(self);
  BODY { /* same code as before */ }
endmethod
```

The `PRE` statement ensures that the *incoming* `mval` is within the expected range while the `POST` statement ensures that `self` is in a valid state *before leaving* the method. Finally,

[34] As the definition in §6.7 of [1] but without *initializer*.

[35] Expression resulting in an object.

the `next_method` call in the `BODY` statement ensures that the contract of `gIncr` is also fulfilled.

Assertions and tests In order to ease the writing of contracts and unit tests, COS provides three standard tests:

- `TestAssert(`*expr*`[,`*file*`,`*line*`])` is a replacement for the standard `assert` and raises an `ExBadAssert` exception on failure. The optional parameters *file* and *line* are transfered to `THROW` when specified.
- `TestType(`*obj*`,`*cls*`[,`*file*`,`*line*`])` is a special case of `TestAssert` which checks for the *exact type* of an object. It is more efficient but less general than `gIsKindOf` and therefore rarely used.
- `TestInvariant(`*obj*`)` checks for the *class invariant* of objects. It can only be used inside methods.

Class invariant The `TestInvariant` assertion relies on the message `gInvariant`. Consequently, it must be specialized for `MilliCounter` in the previous example:

```
defmethod(OBJ, gInvariant, (MilliCounter),
↪        (STR)file, (int)line)
  TestAssert(self->mval < 1000, file, line);
endmethod
```

Here, `TestAssert` propagates the location of the original `TestInvariant` received by `gInvariant` to ease bug tracking. As any other method, `gInvariant` should call its next method (if it exists) to check the invariant defined in its superclasses. In order to avoid unfortunate and useless repetitive invocations of `gInvariant` leading to a complexity of $O(n^2)$, COS doesn't evaluate `TestInvariant` expression when the method is reached through a `next_method` call.

Contracts and inheritance In the design of EIFFEL, Bertrand Meyer recommends to evaluate inherited contracts as a *disjunction* of the preconditions and as a *conjunction* of the postconditions. But [30] demonstrates that EIFFEL-style contracts may introduce behavioral inconsistencies with inheritance, thus COS prefers to treat both pre and post conditions as conjunctions. This is also the only known solution compatible with multimethods where subtyping is superseded by method precedence list.

Contracts level The level of contracts can be set by defining the macro `COS_CONTRACT` to one of:

- `NO` disable contracts (not recommended).
- `COS_CONTRACT_PRE` enables `PRE` statement. This is the recommended level for *production phases*. It is also the default level in COS.
- `COS_CONTRACT_POST` enables `PRE` and `POST` statements. This is the usual level during the *development phases*.
- `COS_CONTRACT_ALL` enables `PRE`, `POST` and `TestInvariant` statements. This is the highest level usually used during *debugging phases* where the potential ineffectiveness of invariants computation doesn't matter.

try-statement:
`TRY`
↪ *statement catch-stmt-list*$_{opt}$ *finally-statement*$_{opt}$
↪ `ENTRY`

catch-stmt-list:
catch-statement
catch-stmt-list catch-statement

catch-statement:
`CATCH(` *class-name* `,` *exception-name* `)` *statement*

finally-statement:
`FINALLY(` *exception-name*$_{opt}$ `)` *statement*

rethrow-statement:
`RETHROW(` *object-expression*$_{opt}$ `);`

throw-statement:
`THROW(` *object-expression* `);`
`THROW(` *object-expression* `,` *file* `,` *line* `);`

exception-name:
identifier

Figure 11. Syntax summary of exceptions.

The contract level is generally the same for the all project but it is also possible to set a different level for each translation unit or even each method.

6. Exceptions

Exceptions are non-local errors which eases the writing of interfaces since they allow to *solve the problems where the solutions exist*. To state it differently, if an *exceptional* condition is detected, the callee needs to return an error and let the caller to take over. Applying recursively this behavior may need to write a lot of code on the callers side to check returned status. Exceptions let the callers choose to either ignore *thrown* errors or to *catch* them and take over.

Implementing an exception mechanism on top of the standard `setjmp` and `longjmp` is not new. But it is uncommon to see a framework written in C which provides the full *try-catch-finally* statements (figure 11) with the same semantic as in other object-oriented programming languages (*e.g.* JAVA, C#). The `CATCH` declaration relies on the message `gIsKindOf` to identify the thrown exception what means that order of `CATCH` definitions matters as in other languages. The `RETHROW(`[*obj*]`)` statement can only be used inside `CATCH` or `FINALLY` definitions. It is worth to know that COS forbids *jump-statement*[35] (including `retmethod`) that would jump

[35] Defined in §6.8.6 in [1].

outside a `TRY`–`ENDTRY` block — *this is an unchecked error!* In practice, it isn't a painful limitation and should favor better programming style.

The sample program below gives an overview of exceptions management in COS:

```
1  #include <cos/Object.h>
2  #include <cos/generics.h>
3  #include <cos/Pointer.h> // for aAllocPointer
4  // standard headers omitted for clarity
5  useclass(AutoRelease,String);
6  useclass(ExBadAssert,ExBadMessage,cExBadAlloc);
7
8  int main(void) {
9    OBJ pool = gNew(AutoRelease);
10   TRY {
11     STR s1 = strdup(''str1''); // not standard
12     OBJ os = aAllocPointer(s1,free); PRT(os);
13     OBJ s2 = gNewWithStr(String,''str2'');PRT(s2);
14
15     TestAssert(os == NIL); // throw ExBadAssert
16
17     UNPRT(os); // unprotect also s2
18     gRelease(os); // equivalent to free(s);
19     gRelease(s2);
20   }
21   CATCH(ExBadAssert, ex) {
22     printf(''assertion %s failed (%s,%d)\n'',
23            gStr(ex), ex_file, ex_line);
24   }
25   CATCH(ExBadMessage, ex) {
26     printf(''unrecognized msg %s sent (%s,%d)\n'',
27            gStr(ex), ex_file, ex_line);
28   }
29   CATCH(cExBadAlloc, ex) { // catch class ExBadAlloc
30     printf(''out of memory (%s,%d)\n'',
31            ex_file, ex_line);
32   }
33   CATCH(Object, ex) { // catch any exception
34     printf(''unexpected exception %s (%s,%d)\n'',
35            gStr(ex), ex_file, ex_line);
36   }
37   FINALLY(ex) { // always executed
38     // ex is the last thrown exception, NIL otherwise
39     if (ex) gRelease(ex);
40   }
41   ENDTRY
42   gRelease(pool);
43 }
```

The code above points out some typical usages:

- Line 9 creates an `AutoRelease` pool because many object factories autorelease newly created objects before returning them. The pool is destroyed at line 42.
- Line 12 creates an automatic `AllocPointer` object to hold and protect `s1` (not the object pointed by `s1`) against exceptions and avoid memory leaks; unless a GC is used.

protection-decl:
`PRT(` *object-decl-list* `);`

unprotection-decl:
`UNPRT(` *object-name* `);`

object-decl-list:
object-name
object-decl-list `,` *object-name*

object-name:
identifier

Figure 12. Syntax summary of protections.

- Line 29 catches the *class* `ExBadAlloc` which is thrown when a memory allocation failure occurs. Throwing an *instance* of the class in a such context would not be safe.
- Line 39 releases the thrown exception. This is also safe in the case of the class `ExBadAlloc` since static objects are insensitive to reference counting.

COS allows to throw *any kind of object* but it provides also a hierarchy of exceptions deriving from `Exception`: `ExBadAlloc`, `ExBadArity`, `ExBadAssert`, `ExBadCast`, `ExBadDomain`, `ExBadFormat`, `ExBadMessage`, `ExBadRange`, `ExBadRetainCount`, `ExBadSize`, `ExBadType`, `ExBadValue`, `ExErrno`, `ExNotFound`, `ExNotImplemented`, `ExNotSupported`, and `ExSignal`. Amongst these exceptions, `ExErrno` and `ExSignal` are special cases used respectively to convert standard *errors* (*i.e.* `TestErrno()`) and *signals* into exceptions.

6.1 Protection

Unless you use a GC or you do not care about resources leakage, you need to *protect and unprotect manually* against exceptions objects with dynamic storage — like in C++ except that *unprotection* is automatic. For this purpose, COS provides (figure 12) the macros `PRT` (push) and `UNPRT` (pop) to manage the *stack*[36] of protected *object* — other resources can be handled with the class `AllocPointer`. If an exception is raised, all the objects protected between the `THROW` and the top-level `TRY-ENDTRY` block receive the message `gRelease`.

7. Closures

Closures are powerful tools when dealing with sequences and containers. The definitions of closure vary from language to language depending on their implementation. To implement function-like objects holding dynamic context, COS provides the family of `gEval{1..5}` messages (equivalent to COMMON LISP `funcall`) and the class cluster `Functor` which implements the mechanism of closure. Unfortunately, C does not support anonymous function

[36] No allocation is performed and no exception can be raised.

but static functions (or generics) can be used in replacement. The objects representing the context of the closure are passed to the `Functor` constructor which supports *incomplete sparse arity* resulting from partial evaluation[37]. The following sample code shows another way to create a counter in PERL using a closure:

```
1  sub counter {
2    my($val) = shift; # seed
3    $cnt = sub { # incr
4      return $val++;
5    };
6    return $cnt; # return the closure
7  }
8
9  $cnt = counter(0);
10 for($i=0; $i<25000000; $i++) {
11   &$cnt();
12 }
```

and can be translated into COS as:

```
1  // includes and useclass omitted for clarity
2
3  static OBJ counter(int seed) {
4    OBJ fct = aFunctor(gIncr,aCounter(seed));
5    return gAutoRelease(fct);
6  }
7
8  int main(void) {
9    OBJ pool = gNew(AutoRelease);
10
11   OBJ cnt = counter(0);
12   for(int i=0; i<25000000; i++)
13     gEval(cnt);
14
15   gRelease(pool);
16 }
```

The line 4 creates the closure using the automatic constructor `aFunctor` which takes the generic function `gIncr` and *deduces* its arity from the remaining parameters, namely the seed *boxed* in the counter. For instance, `fct = aFunctor( gSubTo,,sub)` creates a closure with *arity 2* and `sub` as its *second* argument and the message `gEval2(fct,val)` is equivalent to `gSubTo(val,sub)`. Finally, the message `gAutoRelease` extends the lifespan of both the functor and the counter to the dynamic scope. One can see that COS achieves the same task as PERL with a bit more code but runs more than $\times 15$ *faster*. Moreover, this example highlights a situation where it is safe to use a function instead of the generic `gIncr` since the context is fully known:

```
static OBJ incr(OBJ _1) {
  struct Counter *cnt = (void*)_1; // safe
  ++cnt->val;
  return _1;
}
```

which runs more than $\times 25$ *faster* than PERL.

[37] Sparse arity, composition and curring are beyond the scope of this paper.

8. Design Patterns

It is widely acknowledged that dynamic programming languages simplify significantly the implementation of classical design patterns [26] when they don't superseded them by more powerful dynamic patterns [31, 32, 33]. This section focuses on design patterns enhanced by COS features.

Multiple Inheritance The first version of COS was implementing multiple inheritance using the C3 algorithm [34] to compute the *class precedence list* on the way of DYLAN, PYTHON and PERL6. But it was rapidly considered as too complex for the end-users and incidental as far as fast generic delegation could be achieved. Indeed, multiple inheritance can be simulated by *composition* and *fast generic delegation* with an efficiency close to native support[38] but with more flexibility and less painful design. Combined with dynamic inheritance and multimethods, multiple inheritance by composition and delegation may provide powerful dynamic patterns not yet explored.

Class cluster A class cluster is a set of private classes hidden behind a front-end class which collaborate to implement some states machine. There is numerous examples of class cluster in the literature [35]. The *open object model* allows to define new constructors for the font-end class within the implementation of the hidden classes, leading — leading to better cluster insulation and extensibility — while the *dynamic inheritance* allows to change the class (*i.e.* state) of instances. The CECIL standard library [36] provides numerous examples where class clusters are used as states machine in conjunction with multiple dispatch and predicate dispatch.

High Order Messages Since *high order messages* have already been successfully implemented in OBJECTIVE-C [37], it is not expected to encounter any problem to implement them in COS. Moreover, COS provides the necessary features to simplify the implementation of HOM:

- With fast generic delegation, no need to cache the message in the HOM objects as in the aforementioned paper.
- With multimethods, no need to provide multiple HOM for similar tasks (*i.e.* `gSelect` and `gCollect`).
- With closure supporting sparse arity, no need to construct complex *meta-expressions* or to reorder *compositions*.

HOM are important patterns of modern framework design since they play the role of weavers of *cross-cutting concerns* otherwise solved by foreign technologies based on subject-oriented [38] and aspect-oriented programming [39].

Proxy Proxies have already been discussed previously. But it is worth to say that *fast generic delegation* both strengthens and simplifies a lot this design pattern as well as more than half of the *structural patterns* — in particular the *Decorator* — and some *behavioral patterns*.

[38] OBJECTIVE-C delegation is too slow to achieve multiple inheritance.

Tests	Param.	C++	ObjC	Cos	Cos-*fm*	Cos-*th*
		single dispatch				
`cnt incr`	1	171	58	73	100	97
`cnt incr{1..10}`	1×10	169	60	72	107	98
`cnt incrBy`	$1+1$	154	58	66	87	84
`cnt incrBy2`	$1+2$	154	55	62	89	84
`cnt incrBy3`	$1+3$	128	53	60	78	76
`cnt incrBy4`	$1+4$	124	51	57	78	75
`cnt incrBy5`	$1+5$	115	51	57	72	73
		multiple dispatch				
`cnt addTo`	2	–	–	57	85	85
`cnt addTo2`	3	–	–	53	78	75
`cnt addTo3`	4	–	–	46	69	68
		inheritance				
`mcnt incr`	1	138	55	63	88	87
`mcnt incrBy`	$1+1$	137	53	60	78	76
		generic delegation				
`pxy cnt incr`	1	–	1	39	47	45
`pxy mcnt incr`	1	–	1	37	44	43

Table 3. Performances summary in 10^6 calls/second.

9. Performances

Measuring performances is always a sensitive topic. In order to evaluate the efficiency of Cos, small testsuites have been written to stress the message dispatcher in various conditions. The test results summarized in table 3 have been performed on an Intel(R) Xeon(TM) bi-CPU 2.8 Ghz with the GCC compiler 3.x (and 4.x) to compile the tests written in the three languages into a single program. The timings have been measured with `clock()` and averaged over 10 loops of 10^8 iterations each. The *Param.* column indicates the number of parameters of the message split in *selector-list*+*param-list*. The other columns represent the performances in million of invocations sustained per second for respectively C++ virtual member functions, Objective-C and Cos messages, Cos *fast messages*, and Cos fast messages with Posix *threads* running one thread per CPU. The *Tests* stress the dispatcher with messages already described in this paper: `incr` increments a counter, `incr{1..10}` increments a counter using 10 different methods (to stress the cache), `incrBy{1..5}` accept extra parameters (to stress the `va_list`), `addTo{1..3}` add counters together (to stress multiple dispatch), `mcnt` versions test inheritance and `next_method` calls on millicounter, and finally `pxy` versions test `forward_message` on both counter and millicounter through a proxy. Concerning performances, Cos stays within reasonable ranges since in average it runs about $\times 2.3$ (resp. $\times 1.7$ with fast messages) *slower* than C++ and about $\times 1.2$ (resp. $\times 1.5$ with fast messages) *faster* than Objective-C. Measurements on generic delegation shows that Cos performs $\geq \times 40$ (resp. $\geq \times 50$ with fast messages) *faster* than Objective-C (depending on the hierarchy). Finally, multi-threading does not seem to have a significant impact on performances.

10. Related Work and Conlusion

10.1 Related work

Ooc preprocessor Despite of its age, it is one of the most impressive framework available on this topic [40]. It comes with the `ooc` preprocessor written in Awk. It relies strongly on `void` pointers and requires a lot of manual runtime type checks (*i.e.* `cast(`*cls*`,`*obj*`)`) to ensure correct typing of objects. It gives a full control over inheritance of metaclasses what means that developers can easily break the type system with badly bounded metaclasses. It is one of the first framework for C which uses the concept of generic functions.

Dynace preprocessor This framework has reached the level of commercial product (*not in the public domain*) and comes with the Dynace preprocessor `dpp`, a runtime library, a stable library of classes, and a Windows Development System [41]. Dynace provides features equivalent to those of Objective-C except that it supports multiple inheritance but *not* message forwarding nor exceptions. It comes with its own garbage collector and lightweight threads but it is not as efficient as Objective-C or Cos — message dispatch is about $\times 2$ *slower* even with *jumpTo* assembler code enabled. Moreover, accessing object attributes other than `self` is a bit awkward and relies on specific macros and types (*i.e.* `accessIVs`, `ivPtr`, `ivType`, `GetIVs`, etc...).

Object oriented programming in C This is my previous tentatives over the past five years. One can consult my web page on this topic for more information [42]. In particular, the framework Ooc-2.0 (about 25000 SLOC of C) provides to C89 the *same features* as Java with the same philosophy as Cos (no extra preprocessing). Comparing to Cos, Ooc-2.0 is statically typed and support Java-like dynamic interface. It has also reached the same level of automatic reflection than Java what allowed the fast development of an equivalent JUnit module. But like Java, the static type system makes the design of extensible systems more difficult.

10.2 Future work

Unit testing Cos doesn't have yet a framework for unit tests but already support enough introspection on generics to automate Testcases and Testsuites [43]. It should be one of the next development in order to increase collaboratively with contracts, its reliability across platforms.

Introspection Actually, Cos supports introspection all its components, but not on objects attributes. I plan to extend Cos with features like `deftype` and `defdescriptor` to let the programmer decide what must be known by the reflection. On top of that, the challenge is automatic serialization and encoding of objects for distributed systems. In the mean time, the programmer will have to rely on manually written specialization of `gPut` and `gGet` *a-la*-Boost [44].

Cos ***docgen*** One drawback of the *open object model* is that class interfaces are not defined in a single header

which could serve as documentation. Moreover, multimethods should be documented as part of generics *and* classes. It is planned to write soon an HTML documentation generator on the way of JAVADOC to allow rapid browsing of COS components as well as information on contracts.

COS ***stdlib*** Without a suitable library of classes, COS will be useless. Hence, the development of a library of standard classes on the model of the OBJECTIVE-C Classes Library is a priority. COS *docgen* is probably be a good start for the development of essential classes.

COS ***dbc*** The field of measurement strongly relies on databases to retrieve information on settings, configurations and hardwares descriptions as well as for archiving the results of the data acquisitions and analysis. The experience with OOC-2.0 has shown that dynamic language like COS suit perfectly this kind of development. In particular, dynamic object model and HOM should be the heart of the design of this module.

COS ***script*** The interpretation of scripts to configure and control the application is a domain where COS should provide the necessary features to implement a framework on the way of F-SCRIPT [45]. The figure 13 shows a possible evolution of a project moving towards more flexibility and extensibility while more and more features are added to the application. At a given threshold, the application will need a Domain Specific Language to increase its flexibility or its ease of use. One way to implement a DSL is to use languages like C++ and a lot of code to write a complete interpreter. If the application is highly dynamic and if the parts requiring efficiency are clearly identified and decoupled from the rest, solutions like PYTHON or RUBY with extensions written in C are suitable: the DSL is then a subset of the scripting language. The last solution is to implement the DSL as part of the low-level language like in F-SCRIPT (resp. COS-SCRIPT) where new features are developed as OBJECTIVE-C (resp. COS) classes by programmers and where users have an immediate access to them from the scripting language. This duality between the two languages eases the maintenance of flexible and extensible systems where the DSL just plays the role of the glue between the components and the behaviors. Furthermore, adding new features requires less coding and less knowledge since the same messages and the same data structures are used on both sides leading to no conversion and no efficiency loss. Indeed, once the basic interpreter is written and available itself as a class, nothing more is needed on the side of the DSL as long as the *syntax remains unchanged*. OOPAL is a nice extension example of F-SCRIPT to array programming which highlights the flexibility and the power of this model [46].

10.3 Conclusion

COS seems to be unique (to my knowledge) by the features it provides to the C programming language without requiring a third party preprocessor or compiler, namely: *augmented C syntax to support object oriented programming, uniform object model with extended metaclasses hierarchy, multimethods, fast generic delegation, design by contract, exceptions, closures, and ownership*. This approach allowed to explore rapidly some object models and to select the most appropriate one fulfilling the best the aimed general programming principles: *simplicity, flexibility, extensibility, efficiency* and *portability*. Experiences have shown that full control over the C code is a nice feature by itself and allows to write more reliable code. COS has currently about 7000 SLOC and grows rapidly, but it will take some time before it gets a decent library making it a suitable alternative to existing object oriented programming languages. Moreover, COS has been optimized from the design point of view, but for the sack of simplicity and portability, code tuning has never been performed — a very time consuming task — which lets some room for future improvement. The source code of COS is available on Sourceforge under the LGPL license and the examples of this paper have been extracted from its test suite.

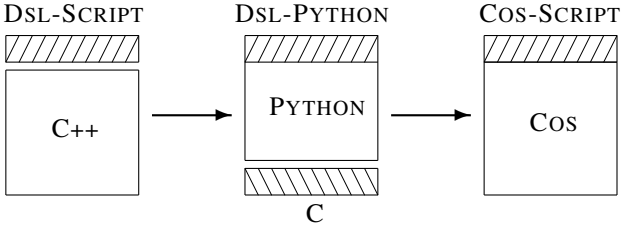

Figure 13. Tradeoff of code productivity, flexibility and extensibility *vs.* efficiency and portability. The conversion of data structures Python ⇔ C can be a source of bottlenecks.

References

[1] International Standard. *Programming Languages – C*. ISO/IEC 9899:1999.

[2] S.P. Harbison, and G.L. Steele. *C: A Reference Manual*. Prentice Hall, 5th Ed., 2002.

[3] J.K. Foderaro. *Lisp is a Chameleon*. Communication of the ACM, Vol. 34 No. 9, 1991.

[4] B.J. Cox, and A.J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[5] A.M. Duncan. *Objective-C Pocket Reference*. O'Reilly, 2003.

[6] Developer Guide. *The Objective-C Programming Language*. Apple Computer Inc., 2006.

[7] S.E. Keene. *Object Oriented Programming in Common Lisp: A Programmers Guide to the Common Lisp Object System*. Addison-Wesley, 1989.

[8] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[9] R.P. Gabriel, J.L. White, and D.G. Bobrow. CLOS*: Integrating Object-Oriented and Functional Programming*. Commu-

nication of the ACM, Vol. 34 No. 9, 1991.

[10] B.W. Kernighan, and R. Pike. *The Practice of Programming*. Addison-Wesley, 1997.

[11] D.R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1997.

[12] J. Bosch. *Design of an Object-Oriented Framework for Measurement Systems*. Systems Domain-Specific Application Frameworks, Ch. 11, John Wiley & Sons, 2000.

[13] J. Bosch, P. Molin, M. Mattsson, and P.O. Bengtsson. *Object-Oriented Framework-based Software Development: Problems and Experiences*. ACM, 2000.

[14] J.M. Nogiec, J. DiMarco, H. Glass, J. Sim, K. Trombly-Freytag, G. Velev and D. Walbridge. *A Flexible and Configurable System to Test Accelarator Magnets*. Particle Accelerator Conference (PAC'01), 2001.

[15] M. Nogiec, J. Di Marco, S. Kotelnikov, K. Trombly-Freytag, D. Walbridge, M. Tartaglia. *A Configurable component-based software system for magnetic field measurements*. IEEE Trans. On Applied Superconductivity, Vol. 16 No. 2, 2006.

[16] P. Arpaia, L. Bottura, M. Buzio, D. Della Ratta, L. Deniau, V. Inglese, G. Spiezia, S. Tiso, L. Walckiers. *A software framework for flexible magnetic measurements at CERN*. Instrumentation and Measurement Technology Conference (IMTC'07), 2007.

[17] J.C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2001.

[18] R.E. Johnson. *Dynamic Object Model*. `http://st-www.cs.uiuc.edu/users/johnson/papers/dom`, 1998

[19] D. Riehle, M. Tilman and R.E. Johnson. *Dynamic Object Model*. 7th Conference on Pattern Languages of Programs (PLoP'2000), 2000.

[20] J.W. Yoder and R.E. Johnson. *The Adaptive Object-Model Architectural Style*. 3rd Working IEEE Conference on Software Architecture (WICSA'2002), 2002.

[21] J. van Gurp, and J. Bosch. *Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines*. Software Practice & Experience, John Wiley & Sons, March 2001.

[22] D. Parsons, A. Rashid, A. Telea, and A. Speck. *An architectural pattern for designing component-based application frameworks*. Software Practice & Experience, John Wiley & Sons, November 2005.

[23] H. Boehm. *Bounding Space Usage of Conservative Garbage Collectors*. 30th ACM Sigplan Symposium on Principles of Programming Languages (PoPL'2002). `http://www.hpl.hp.com/personal/Hans_Boehm/gc`.

[24] R. Razavi, N. Bouraqadi, J. Yoder, J.F. Perrot, and R. Johnson. *Language support for Adaptive Object-Models using Metaclasses*. ESUG Conference 2004 Research Track, 2004.

[25] N.M. Bouraqadi-Saâdani, T. Ledoux, and F. Rivard. *Safe Metaclass Programming*. OOPSLA'98, 1998.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[27] E. Dujardin, E. Amiel and E. Simon. *Fast Algorithm for Compressed Multimethod Dispatch Table Generation*. ACM Transactions on Programming Languages and Systems, Vol. 20, January 1998.

[28] Y. Zibin and Y. Gil. *Fast Algorithm for Creating Space Efficient Dispatching Table with Application to Multi-Dispatching*. OOPSLA'02, 2002.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd Ed., 1997.

[30] R.B. Findler, M. Latendresse, and M. Felleisen. *Behavioral Contracts and Behavioral Subtyping*. 9th ACM Sigsoft International Symposium on Foundations of Software Engineering, 2001.

[31] P. Norvig. *Design Patterns in Dynamic Programming*. `http://www.norvig.com/design-patterns`, 1996.

[32] G.T. Sullivan *Advanced Programming Language Features for Executable Design Pattern*. Technical Report AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

[33] Developer Guide. *Cocoa Fundamentals Guide*. Apple Computer Inc., 2006.

[34] K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford, and P. Tucker Withington. *A monotonic superclass linearization for Dylan.* OOPSLA'96, 1996.

[35] Developer Guide. *Foundation Framework Reference*. Apple Computer Inc., 2006.

[36] The Cecil Group. *Cecil Standard Library Manual*. Department of Computer Science and Engineering, University of Washington, 2004.

[37] M. Weiher and S. Ducasse. *High Order Message*. OOPSLA'05, 2005.

[38] W. Harrison, H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects)*. OOPSLA'93, 1993.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. European Conference on Object-Oriented Programming (ECOOP'97), 1997.

[40] A.T. Schreiner. *Object Oriented Programming with ANSI C*. `http://www.planetpdf.com/codecuts/pdfs/ooc.pdf`, 1994.

[41] B. McBride. *Dynace: Dynamic C language extension*. `http://algorithms.us/`, 2006.

[42] L. Deniau. *Object Oriented Programming in C*, 2001–2006. `http://cern.ch/laurent.deniau/oopc.html`.

[43] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[44] C++ community. *Boost C++ Library*. http://www.boost.org.

[45] P. Mougin. *The F-Script Language*, 1998–2006. `http://www.fscript.org/`.

[46] P. Mougin, and S. Ducasse. *OOPAL: Integrating Array Programming in Object-Oriented Programming*. OOPSLA'03, 2003.