The C Object System*

Using C as a High-Level Language

Laurent Deniau
CERN – European Organization for Nuclear Research
CH-1211 Geneva, Switzerland
laurent.deniau@cern.ch

ABSTRACT

The C Object System (COS) is a pure C library which implements high-level concepts available in CLOS, OBJECTIVE-C and other programming languages: *uniform object model* (class, metaclass and property-metaclass), *generic functions, multi-methods, delegation, properties, exceptions, contracts* and *closures*. COS relies on the programmable capabilities of the C programming language to extend its syntax and implement the aforementioned concepts as *first-class objects*. COS aims at satisfying seven general principles all together: *simplicity, flexibility, extensibility, reusability, variability, efficiency* and *portability*. Its inductive design is tuned to provide efficient and portable implementation of two key concepts essential to support these principles: *message multi-dispatch* and *message multi-forwarding*. With COS features at hand, software becomes as simple and flexible as with scripting languages and as efficient and portable as expected with C programming. Likewise, COS concepts significantly simplify *adaptive* and *dynamic* programming as well as *large-scale* and *service-oriented* software development.

Categories and Subject Descriptors

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

General Terms

Dynamic Languages

Keywords

Adaptive object model, Aspects, Class cluster, Closure, Contract, Delegation, Design pattern, Exception, Generic function, Introspection, High-order message, Message forwarding, Meta-class, Meta-object protocol, Multi-method, Open class, Predicate dispatch, Properties, Uniform object model.

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

1. INTRODUCTION

The C Object System (COS) is a pure C library which adds powerful concepts with user-friendly syntax to the C programming language [1, 2] using its *programmable capabilities*[1]. COS has been developed in order to solve fundamental programming problems encountered by tiny to medium sized teams in software development by enhancing above all the simplicity, efficiency and variability of design and code.

Every year, new development models, design models, or programming languages are invented and promoted. They all claim to solve the problems encountered in other existing model or programming languages. For instance, *dynamically typed languages* focus on simplicity and expressivity, and promote adaptive and dynamic programming as well as rapid development, while *statically typed languages* focus on compile-time computations (including type safety) and runtime efficiency. To have loose coupling, the latter must rely on dynamic features like JAVA and C# interfaces or OBJECTIVE-C protocols. More sophisticated languages further support collaborative features (*i.e.* roles) like SCALA traits, HASKELL classes and C++ parametrized inheritance, all of which allow defining and composing *mixins*. But these features solve only part of the problem and require explicit collaboration between complex type systems and runtime polymorphism. Besides, they also have to rely on various development models and design recipes to improve the flexibility of the code and sustain large-scale development.

In contrast, COS relies on the C programming language which is *simple*, widely known and used, and has an abstract machine close to the physical machine. This allows translating *portable* code into very *efficient* and predictable executables and to apply simple development models. The drawback is that COS cannot raise the expressive power of C itself to build domain specific languages or perform meta-programming and generative programming like in C++.

1.1 From Principles ...

COS aims at satisfying seven general principles rarely met all together in a single programming language — *simplicity, flexibility, extensibility, reusability, variability, efficiency and portability* — which have thus guided the design.

Simplicity The language must be easy to learn and use. The training curve for an average programmer should be

[1]As in "*Lisp is a programmable programming language*" [3].

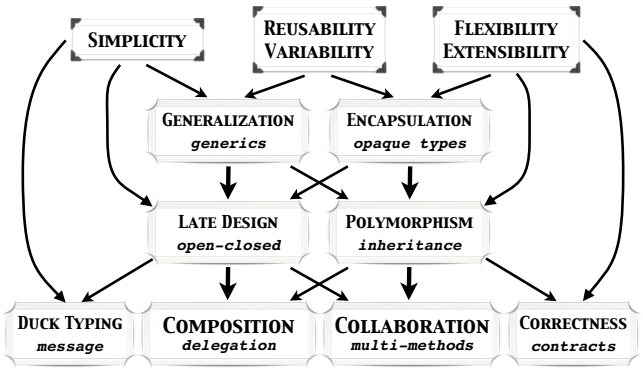

Figure 1: From principles to concepts.

as short as possible what implies a clear and concise syntax. Simplicity should become an asset which guarantees the quality of the code and allows writing complex constructions without being penalized by a complex formalism or by the multiplicity of the paradigms. COS can be learned in a few days by experienced C programmers, although exploiting the full power of COS requires some training.

Flexibility and Extensibility The language must support the improvement of existing *components* and the addition of new features without changing significantly their design.

Reusability and Variability The language must support the ability to reuse or quickly adapt existing *components* to unforeseen tasks and to *compose* them dynamically.

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* resource usage. It is also an asset for simplicity when "*brute force is enough*".

Portability A general purpose programming language must be portable, that is it must be widely available on many architectures and accessible from other languages (FFI). This 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.

1.2 ... to Concepts

COS extends the C programming language with concepts borrowed from OBJECTIVE-C [4, 5] and CLOS [7, 8] and implemented in a unique collaborative and efficient manner. Its entire design focuses on two fundamental concepts:

- *dynamic polymorphic collaboration* (*i.e.* multi-method)
- *dynamic polymorphic composition* (*i.e.* delegation)

Figure 1 shows the main inductive paths from the aimed principles to the required concepts which guided the entire design and implementation of COS and formally described hereafter.

Encapsulation and Generalization Encapsulation is a major concern when developing libraries and large-scale software because encapsulation is not only a matter of loose coupling but also a design issue. Hence, COS objects always have *opaque types* (*i.e.* `OBJ`) to ensure strong encapsulation of implementations. Moreover object behaviors are represented by generic functions which enforce the *separation of concerns* since they generalize interfaces [17] towards single orthogonal and generic open multi-methods [24] (section 3).

Polymorphism and Late Design Extending components (*i.e. reusable black boxes*) requires inheritance and polymorphism to postpone at runtime the resolution of methods invocation. Besides, when the resolution is achieved by *message dispatch*, the coupling between callers and callees becomes almost inexistent and the code size and complexity are significantly reduced. Furthermore, the *open class model* allows extending classes *on need* by adding new methods without breaking the encapsulation (*i.e.* without "*reopening the box*") and thus reduces the risk of premature design. On one hand, these simplifications improve the programmer understanding who makes less conceptual errors, draws simpler design and increases its productivity. On the other hand, message dispatch postpones the detection of unknown behavior at runtime, with the risk to see programs ending abnormally, but *contracts* and test suites restrict this risk to exceptional situations.

Collaboration Software development is mainly about building collaborations between entities, namely objects. As soon as polymorphic objects are involved everywhere to ensure good software extensibility and reusability, one needs *polymorphic collaboration* implemented by *multi-methods* [24]. They reduce strong static coupling that exist in the Visitor pattern (or equivalent) as well as the amount of code needed to achieve the task. COS performs message multi-dispatch with an efficiency equivalent to C++ virtual member function (sections 4 & 5).

Composition The composition of objects and behaviors is the cornerstone of software flexibility and variability through the use of indirections. Most structural and behavioral design patterns described in [22] introduce such indirections, but they also increase the code complexity and rigidity and hence decrease the reusability of the components built. The *delegation* is an effective mechanism which allows managing *dynamic composition* of both, objects and behaviors, without introducing coupling. COS achieves delegation with the same efficiency as message dispatch, *seemingly a unique feature* (section 4.2).

Reflection Reflection is a powerful aspect of adaptive object models which, amongst others, allows to mimic the behavior of interpreters. COS provides full introspection and limited intercession on polymorphic types and behaviors, that is classes, generics and methods, as well as object attributes through the definition of properties. Since all COS components are *first-class objects*, it is trivial to replace creational patterns [22] by generic functions (section 6.1).

Ownership The management of object life cycles requires a clear policy of ownership and scope rules. 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 choose between garbage collection [19] and *(semi-)*

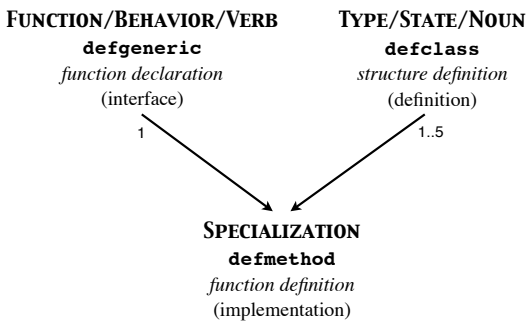

Figure 2: COS **components mapping to C-forms**

manual reference counting with rich semantic (section 2.3).

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

Consequently, the concepts resulting from the inductive design of COS strengthen inherently all the guidelines stated in [17] to build "*flexible, usable and reusable object-oriented frameworks*" as well as the architectural pattern proposed in [18] to design *flexible component-based frameworks*.

1.3 Components

COS is a small framework entirely written in portable[2] ISO C which provides programming paradigms like *objects, classes, metaclasses, generic functions, multi-methods, delegation, properties, exceptions, contracts* and *closures*. COS syntax and features are directly available at the level of the C source code through the use of the language keywords defined in the header file `<cos/Object.h>`. The object-oriented layer of COS is based on three components (figure 2) borrowed from CLOS which characterize its *meta-object protocol* [7, 8].

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

Generics Generics play the same role as *function declarations* in C and define *messages*. They are essential actors to ensure good separation of concern and correctness of formal parameters of messages between callers and callees.

Methods Methods play the same role as *function definitions* in C and define *specializations* of generics. A method is invoked if the message belongs to its generic and the receivers match its classes (multi-methods).

The similarity between the COS components and their equivalent C-forms (figure 2) actually makes simple procedural development model very effective with COS and lets C programmers be rapidly productive compared to other object-oriented approaches which impose more constraints. This is a consequence of the *open class model* which enables adding methods on need in different places while affording a much better extensibility and reusability of the components built.

[2]Namely C89 augmented with variadic macros (C99).

Syntax COS introduces new keywords to extend the C language with a user-friendly syntax half-way between OBJECTIVE-C and CLOS. COS *parses its syntax and generates code with the help of its functional C preprocessing library*[3]; a module of a few hundred C macros which was developed for this purpose. It offers limited parsing capabilities, token recognition, token manipulation and algorithms like *eval, map, filter, fold, scan, split* borrowed from functional languages and working on tuples of cpp-tokens. As a rule of thumb, all COS symbols and macros are mangled to avoid unexpected collisions with other libraries, including the language keywords which can be disabled individually.

Despite of its dynamic nature, COS tries hard to detect all syntax errors, type errors and other mistakes at compile-time by using *static asserts* or similar tricks and to emit clear and meaningful diagnostics. The only thing that COS cannot check at compile time is the understanding of a message by its receivers; an important "feature" to reduce coupling.

2. CLASSES *(NOUNS, ENTITIES)*

COS allows defining and using classes as easily as in other object-oriented programming languages.

Declaration The `useclass()` *declaration* allows accessing classes as *first-class objects*. The following simple program highlights the similarities between COS and OBJECTIVE-C:

```
1  #include <cos/Object.h>
2  #include <cos/generics.h>
3
4  useclass(Counter, (Stdout)out);
5
6  int main(void) {
7    OBJ cnt = gnew(Counter);
8    gput(out, cnt);
9    gdelete(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, Stdout;
5
6  int main(void) {
7    id cnt = [Counter new];
8    [Stdout put: cnt];
9    [cnt release];
10 }
```

Line 2 makes the standard generics like `gnew`, `gput` and `gdelete`[4] visible in the current translation unit. OBJECTIVE-C doesn't need this information since methods are bound to their class, but if the user wants to be warned for incorrect use 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 fine tuning of exposure of interfaces since only the used generic functions have to be visible.

Line 4 declares the class `Counter`[5] and the alias `out` for local renaming of the class `Stdout`, both classes being supposedly

[3]Describing this module is beyond the scope of this paper.
[4]By convention, COS generic names starts by a '`g`'.
[5]By convention, COS class names starts by an uppercase.

defined elsewhere to avoid an error at link-time. In line 7, the generic type `OBJ` is equivalent to `id` in Objective-C.

Lines 7–9 show the life cycle of objects, starting with `gnew` (resp. `new`) and ending with `gdelete` (resp. `release`). They also show that generics *are* functions (*e.g.* one can take their address). Finally, the line 8 shows an example of multi-method where the message `gput(_,_)` will look for the specialization `gput(mStdout,Counter)` whose meaning is discussed in section 4. In order to achieve the same task, Objective-C code has to rely on the Visitor pattern, a burden that requires more coding, creates static dependencies (strong coupling) and is difficult to extend.

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

```
defclass(Counter)
  int cnt;
endclass
```

which is translated in Objective-C as:

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

or equivalently in Clos as:

```
(defclass Counter (Object) ((cnt)) )
```

The `Counter` class derives from the root class `Object` — the default behavior when the superclass isn't specified — and defines the attribute `cnt`.

Visibility *What must be visible and when?* In order to manage coupling, Cos provides three levels of visibility: none, declaration and definition. If only the generic type `OBJ` is used, nothing is required *(no coupling)* as in:

```
OBJ gnew(OBJ cls) {
  return ginit(galloc(cls));
}
```

If instances of a class are created, only the declaration is required *(weak coupling)* as in:

```
OBJ gnewBook(void) {
  useclass(Book); // declaration of a local reference
  return gnew(Book);
}
```

If subclasses, methods or instances with automatic storage duration are defined, the class definition (*i.e.* `defclass`) must be visible *(strong coupling)*.

2.1 Inheritance

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

```
defclass(MilliCounter, Counter)
  int mcnt;
endclass
```

which gives in Objective-C:

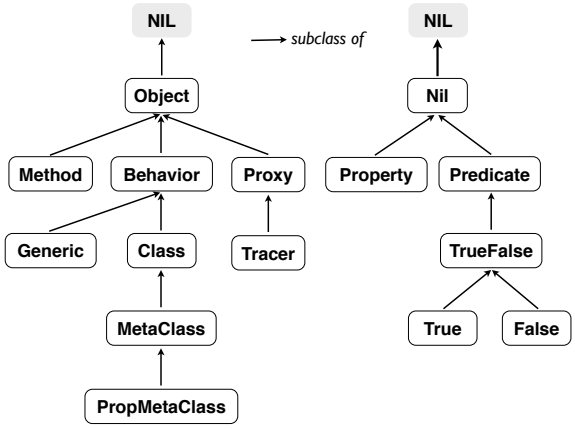

Figure 3: Subset of Cos core classes hierarchy

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

and in Clos:

```
(defclass MilliCounter (Counter) ((mcnt)) )
```

In the three cases, the derived class inherits the attributes and the methods of its superclass. 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 behaviors in the context of *multi-methods* is discussed in section 4 and the emulation of multiple inheritance in section 6.2.

Class rank Cos computes at compile-time the inheritance depth of each class. The rank of a root class is zero (by definition) and each successive subclass increases the rank.

Dynamic inheritance Cos provides the message `gchangeClass(`*obj*`,`*cls*`)` to change the class of *obj* to *cls* iff it is a superclass of *obj*'s class; and the message `gunsafeChangeClass(`*obj*`,`*cls*`,`*spr*`)` to change the class of *obj* to *cls* iff both classes share a common superclass *spr* and the instance size of *cls* is lesser or equal to the size of *obj*. These messages are useful for implementing *class clusters*, *state machines* and *adaptive behaviors*.

2.2 Meta classes

Like in Objective-C, a Cos class definition creates a parallel hierarchy of metaclass which facilitates the use of *classes as first-class objects*. Figure 4 shows the complete hierarchy of the `PropMetaClass` class, including its metaclasses.

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 [20]. Both inheritance trees are built in parallel: if a class `A` derives from a class `B`, then its metaclass `mA`[6] derives from the metaclass `mB` — except the root classes which derive from `NIL` and have their metaclasses deriving from `Class` to *close the inheritance path*. All meta-classes are instances of the class `MetaClass`.

[6]Metaclass names are class names prefixed by '`m`'.

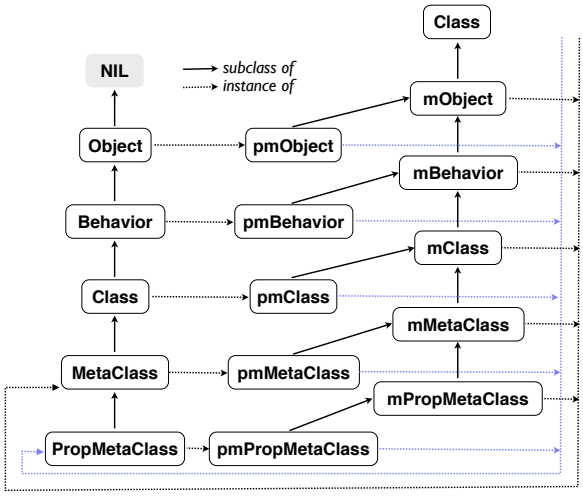

Figure 4: Subset of Cos core classes with metaclasses

Property metaclass In some design patterns like Singleton or Class Cluster, or during class initialization, the automatic derivation of the class metaclass from its superclass metaclass can be problematic as detailed in [21]. To solve the problem Cos associates to 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, namely `pmA`[7] (resp. `pmB`) derives from `mA` (resp. `mB`) as shown in the figure 4. All property meta-classes are instances of the class `PropMetaClass`.

Class objects With multi-methods and metaclasses in hands, it is possible to use classes as common objects. Figure 5 shows the hierarchy of the core class-objects used in Cos to specialized multi-methods with specific *states*. For instance messages like `gand`, `gor` and `gnot` are able to respond to messages containing the class-predicates `True`, `False` and `TrueFalse` (*i.e.* undefined). The root class `Nil` is a special class-object which means *no-object* but still safe for message dispatch: sending a message to `Nil` is safe, but not to `NIL`.

Type system The Cos type system follows the rules of Objective-C, that is *polymorphic objects have opaque types (ADT) outside their methods and are statically and strongly typed inside their methods*; not to mention that multi-methods reduce significantly the need for runtime identification of polymorphic parameters. Furthermore, the set of class–metaclass–property-metaclass forms a *coherent hierarchy of classes and types* which offers better consistency and more flexibility than in Objective-C and Smalltalk where metaclasses are not explicit and derive directly from `Object`.

2.3 Instances

Object life cycle The life cycle of objects in Cos *starts* by the creation (`galloc`) followed by the initialization (`ginit` and variants) and *ends* with the deinitialization (`gdeinit`) followed by the destruction (`gdealloc`). In between, the user manages the ownership of objects (*i.e.* dynamic scope)

[7]Property metaclass names are class names prefixed by '`pm`'.

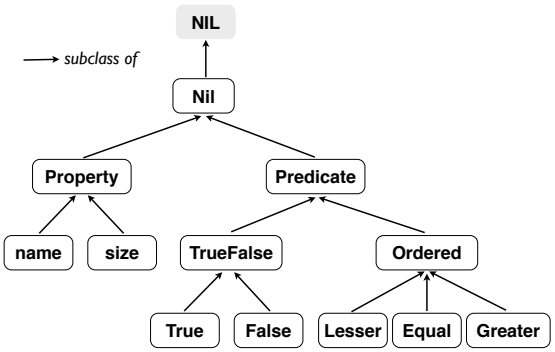

Figure 5: Subset of Cos class-predicates

with `gretain`, `grelease` and `gautoRelease` like in Objective-C. The *copy initializer* is the specialization of the generic `ginitWith(_,_)` for the same class twice. The *designated initializer* is the initializer with the most coverage which invokes the designated initializer of the superclass using `next_method`. Other initializers are *secondary initializers* which must invoke the designated initializer [6].

Object type In Cos (resp. Objective-C), objects are always of dynamic type because the type of `galloc` (resp. `alloc`) is `OBJ` (resp. `id`). Since it is the first step in the life cycle of objects in both languages, the type of objects can never be known statically, except inside their own multi-methods. That is why Cos (resp. Objective-C) provides the message `gisKindOf(`*obj*`,`*cls*`)` (resp. `[`*obj* `isKindOf:` *cls*`]`) to inspect the type of objects. But even so, it would be dangerous to use a static cast to convert an object into its expected type because dynamic design patterns like Class Cluster and Proxy might override `gisKindOf` for their own use. Cos also provides the message `gclass(`*obj*`)` which returns *obj*'s class.

Object identity In Cos, an object is bounded to its class through a unique 32-bit identifier produced by a linear congruential generator which is also a generator of the cyclic groups $\mathbb{N}/2^k\mathbb{N}$ for $k = 2..32$. This powerful algebraic property allows retrieving efficiently the class of an object from the components table using its identifier as an index (Figure 6). Comparing to pointer-based implementations, the unique identifier has four advantages:

It ensures better behavior of cache lookups under heavy load (uniform hash), it makes the hash functions very fast (sum of shifted `ids`*), it is smaller than pointers on 64-bit machines and it can store extra information (high bits) like class ranks to speedup linear lookup in class hierarchies.*

Automatic objects Since Cos adds an object-oriented layer on top of the C programming language, it is possible to create objects with automatic storage duration (*e.g.* on the stack) using compound literals. In order to achieve this, the class definition must be visible and the developer of the class must provide a special constructor. For example the constructor `aStr("a string")`[8] is equivalent to the Objective-C directive `@"a string"`. Cos already provides automatic constructors for many common objects like `Char`, `Short`, `Int`, `Long`, `Float`, `Complex`, `Range`, `Functor` and `Array`. *Automatic*

[8]By convention, *automatic* constructors start by an 'a'.

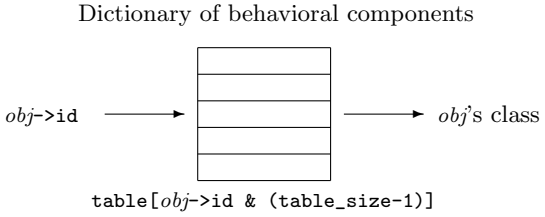

Figure 6: Lookup of object's class from object's id.

constructors allow creating efficiently temporary objects with local scope and enhance the flexibility of multi-methods. For example, the copy initializer `ginitWith(_,_)` and its variants can be used in conjunction with almost all the automatic constructors aforementioned. Thanks to the rich semantic of COS reference counting, if an automatic object receives the message `gretain`, `gautoDelete` or `gautoRelease`, it is automatically cloned using the message `gclone` 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 they require some care in multi-threaded environments. It is worth noting that all COS *components* have static storage duration and consequently are *insensitive to ownership and destruction.*

3. GENERICS *(VERBS, ROLES)*

We have already seen in previous code samples that generics are C functions. But a generic is in fact a verb raised at the same level of abstraction as a noun and takes multiple forms: i) a *function declaration* (`defgeneric`) which ensures the correctness of the signature of its methods (`defmethod`), aliases (`defalias`) and next-methods (`defnext`), ii) a *function definition* used to dispatch the message and find the most specialized method belonging to the generic and matching the classes of the *receivers*, iii) an *object* holding the generic's metadata: *the selector.*

Generic rank The rank of a generic is the number of receivers in its list of formal parameters. COS supports generics from rank 1 to 5 what should be enough in practice since rank 1 to 4 already cover all the multi-methods defined in the libraries of CECIL and DYLAN [23, 30].

Declaration Generic declarations are less common than class declarations but they can be useful when one wants to use generics as first-class objects. Since generic definitions are more often visible than class definitions, it is common to rename them locally as in the following short example:

```
void safe_print(OBJ obj) {
  usegeneric( (gprint) prn );
  if ( gunderstandMessage1(obj, 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];
}
```

Definition Definition of generics correspond to function declarations in C and differ from OBJECTIVE-C method declarations by the fact that they are neither bound to classes (prefix '-') nor to metaclasses (prefix '+'). The following definitions:

```
defgeneric(void, gincr, _1);                    // rank 1
defgeneric(void, gincrBy, _1, (int)by);         // rank 1
defgeneric(OBJ , ginitWith, _1, _2);            // rank 2
defgeneric(OBJ , ggetAt, _1, at);               // rank 2
defgeneric(void, gputAt, _1, at, what);         // rank 3
```

can be translated into CLOS (but not OBJECTIVE-C) as:

```
(defgeneric incr (obj))
(defgeneric incr-by (obj by))
(defgeneric init-with (obj with))
(defgeneric get-at (obj at))
(defgeneric put-at (obj at what))
```

Selector parameters like `at` are called *open types* (no parenthesis) since their type can vary for each specialization. Other parameters like `by` are called *closed types* (with parenthesis) and have fixed types and names: specializations must use the same types and names as defined by the generic. This enforces the semantic of *monomorphic* parameters which could otherwise be ambiguous: `(int)offset` *vs.* `(int)index`.

Messages COS dispatcher uses global caches (one per generics rank) implemented with hash tables to speedup method lookups. The caches solve slot collisions by growing until they reach a configurable upper bound of slots. After that, they use sorted linked lists to hold a maximum of 3 cells. Above this length, the caches start to forget cached methods — a required behavior when dynamic class creation is supported. The lookup uses *fast asymmetric hash functions* (sum of shifted `id`s) to compute the cache slots and ensures uniform distribution even when all selectors have the same type or specializations on permutations exist.

4. METHODS

Methods are defined using a similar syntax as generics as shown in the following code which defines a method specialization of the generic `gincr` for the class `Counter`:

```
defmethod(void, gincr, Counter)
  self->cnt++;
endmethod
```

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

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

Methods specializers The *receivers* can be equivalently accessed through `self`n[9] whose types correspond to their class specialization (*e.g.* `struct Counter*`) and through unnamed parameters `_`n whose types are `OBJ` for $1 \leq n \leq g$, where g is the rank of the generic. It is important to understand that `self`n and `_`n are bound to the same object, but `self`n provides a statically typed view (*i.e.* monomorphic) which allows *treating efficiently* COS *objects as normal C structures.*

Multi-methods Multi-methods are methods with more than one receiver and do not require special attention in COS. The following example defines the assign-sum operator (*i.e.* `+=`) specializations which add respectively 2 and 3 `Counters`:

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

```
defmethod(OBJ, gaddTo, Counter, Counter)
  self->cnt += self2->cnt;
  retmethod(_1); // return self
endmethod

defmethod(OBJ, gaddTo2, Counter, Counter, Counter)
  self->cnt += self2->cnt + self3->cnt;
  retmethod(_1); // return self
endmethod
```

In effect, about 75% of Cos *generics have a rank > 1 (multi-methods) and cover more than 90% of all* Cos *methods.*

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

```
defmethod(void, ginitialize, pmMyClass)
  // some initialization specific to MyClass when it's loaded.
endmethod

// full implementation of the 9 states tri−boolean table
defmethod(OBJ, gand, mFalse, mTrueFalse)
  retmethod(_1); // return False
endmethod

defmethod(OBJ, gand, mTrueFalse, mFalse)
  retmethod(_2); // ditto
endmethod

defmethod(OBJ, gand, mTrueFalse, mTrueFalse)
  retmethod(TrueFalse);
endmethod

defmethod(OBJ, gand, mTrue, mTrue)
  retmethod(_1); // return True
endmethod
```

Method aliases Cos allows specializing compatible generics with the same implementation. The following aliases define specializations for `gpush`, `gtop` and `gpop` which share the specializations of `gput`, `gget` and `gdrop` respectively:

```
defalias(void, (gput )gpush, Stack, Object);
defalias(OBJ , (gget )gtop , Stack, Object);
defalias(void, (gdrop)gpop , Stack, Object);
```

Method types In order to support fast generic delegation (section 4.2), Cos must use internally the same function types (*i.e.* same C function signatures) for methods implementation belonging to generics of the same rank:

```
void (*IMP1)(SEL,OBJ,void*,void*);
void (*IMP2)(SEL,OBJ,OBJ,void*,void*);
void (*IMP3)(SEL,OBJ,OBJ,OBJ,void*,void*);
void (*IMP4)(SEL,OBJ,OBJ,OBJ,OBJ,void*,void*);
void (*IMP5)(SEL,OBJ,OBJ,OBJ,OBJ,OBJ,void*,void*);
```

The first parameter `_sel` and the subsequent `OBJ`s point respectively to the message selector (*i.e.* generic's object) and to the receivers (*i.e.* `_n`) used by the dispatcher, the penultimate parameter `_arg` is a pointer to the structure storing the closed arguments of the generic (if any) and the last parameter `_ret` is a pointer to the placeholder of the returned value (if any). The responsibilities are shared as follow:

i) the *generic functions* are in charge to pack the closed arguments into the structure pointed by `_arg`, to create the placeholder pointed by `_ret` for the returned value, to lookup for the method specialization and to invoke its implementation (*i.e.* `IMP`*n*) with the arguments `_sel`, `_n`, `_arg` and `_ret`.

ii) the *methods* are in charge to unpack the closed arguments into local variables, to bind the receivers `_n` to local `self`*n* and to handle the returned value appropriately.

4.1 Next method

The `next_method` principle borrowed from Clos[10] is an elegant answer to the problem of superclass(es) methods *call* (*i.e.* late binding) in the presence of *multi-methods*. The following sample code defines a specialization of the message `gincrBy` for the class `MilliCounter` which adds thousandths of count to the class `Counter`:

```
1 defmethod(void, gincrBy, MilliCounter, (int)by)
2   self->mcnt += by;
3   if (self->mcnt >= 1000) {
4     defnext(void, gincr, MilliCounter);
5     self->mcnt -= 1000;
6     next_method(self); // call gincr(Counter)
7   }
8 endmethod
```

which is equivalent to the Objective-C code:

```
- (void) incrBy: (int)by {
  self->mcnt += by;
  if (self->mcnt >= 1000) {
    self->mcnt -= 1000;
    [super incr];
  }
}
```

Line 6 shows how Cos `next_method` replaces the message sent to `super` in Objective-C. By default, `next_method` calls the next method belonging to the same generic (*e.g.* `gincrBy`) where *next* means the method with the highest specialization less than the current method. But in the example above, the `Counter` class has no specialization for `gincrBy`. That is why the line 4 specifies an *alternate next method path*, namely `gincr`, to redirect the `next_method` call to the appropriate next method. In some cases, it might be safer to test for the existence of the next method before calling it:

```
if (next_method_p) next_method(self);
```

It is worth noting that `next_method` transfers the returned value (if any) directly from the called next method to the method caller. Nevertheless, the returned value can still be accessed through the *lvalue* `RETVAL`.

Methods specialization Assuming for instance the class inheritance `A >: B >: C`, the *class precedence list* for all pairs of specialization of `A`, `B` and `C` by *decreasing order* is:

```
(C,C)(C,B)(B,C)(C,A)(B,B)(A,C)(B,A)(A,B)(A,A)
```

and the list of *all* `next_method` *paths* are:

```
(C,C)(C,B)(C,A)(B,A)(A,A)
(B,C)(B,B)(B,A)(A,A)
(A,C)(A,B)(A,A)
```

The algorithm used by Cos to build the class precedence list (*i.e.* compute method rank) has some nice properties: it provides natural asymmetric *left-to-right precedence* and it is *non-ambiguous*, *monotonic* and *totally ordered* [29].

Around methods Around methods borrowed from Clos provide an elegant mechanism to enclose the behavior of some *primary method* by an arbitrary number of around methods. Around methods are always *more specialized* than their primary method but have an undefined precedence:

```
defmethod(void, gdoIt, A, A)
endmethod
```

[10] Namely `call-next-method`.

```
defmethod(void, gdoIt, B, A)
  next_method(self1, self2); // call gdoIt(A,A)
endmethod

defmethod(void, (gdoIt), B, A) // around method
  // pre−processing
  next_method(self1, self2); // call gdoIt(B,A)
  // post−processing
endmethod

defmethod(void, gdoIt, B, B)
  next_method(self1, self2); // call (gdoIt)(B,A)
endmethod
```

4.2 Delegation

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

Unrecognized message Message dispatch performs runtime lookup to search for method specializations. If no specialization is found, the message `gunrecognizedMessage`n is SUBSTITUTED and *sent with the same arguments as the original sending, including the selector*. Hence these messages can be overridden to support the delegation or implement some adaptive behaviors. By default, `gunrecognizedMessage`n throws the exception `ExBadMessage`.

Forwarding message Message forwarding has been borrowed from OBJECTIVE-C and extended to multi-methods. The code sample below shows a common usage of message forwarding to protect objects against invalid messages:

```
1 defmethod(void, gunrecognizedMessage1, MyProxy)
2   if(gundertstandMessage1(self->obj,_sel) == True)
3     forward_message(self->obj); // delegate
4 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 }
```

Here, `forward_message` propagates all the arguments, including the hidden parameters `_sel`, `_arg` and `_ret`, to a different receiver. As for `next_method`, `forward_message` transfers the returned value directly to the method caller and can still be accessed through `RETVAL` in the same way.

Fast delegation Since all methods belonging to generics with equal rank have the same C function signature and fall into the same lookup cache, it is safe to cache the message `gunrecognizedMessage`n in place of the *unrecognized message* and the next sending of the latter will result in a cache hit.

> *This substitution allows the delegation to be as fast as message dispatch, seemingly a unique feature.*

Intercession of forwarded messages Since the closed arguments of the generic's parameters are managed by a C structure, it is possible to access each argument separately. In order to do this, COS provides introspective information on generics (*i.e.* metadata on types and signatures) which allows identifying and retrieving the arguments and the returned value efficiently. But this kind of needs should be exceptional and is beyond the scope of this paper.

4.3 Contracts

To quote Bertrand Meyer [25], 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, and the callee must meet all postconditions — the failure of either party leads to a bug in the software. To illustrate how contracts work in COS, we can rewrite the method `gincr`:

```
defmethod(void, gincr, Counter)
  int old_cnt;
  PRE old_cnt = self->cnt;
  POST test_assert(self->cnt > old_cnt);
  BODY self->cnt++;
endmethod
```

The `POST` statement `test_assert` checks for counter overflow *after* the execution of the `BODY` statement and throws the exception `ExBadAssert` on failure, breaking the contract. 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. As well, `gincrBy` can be improved:

```
defmethod(void, gincrBy, MilliCounter, (int)by)
  PRE test_assert(by >= 0 && by < 1000, "out or range");
  BODY // same code as before
endmethod
```

The `PRE` statement ensures that the *incoming* `by` is within the expected range and the `next_method` call in the `BODY` statement ensures that the contract of `gincr` is also fulfilled.

Assertions and Invariants The `test_assert` is a replacement for the standard `assert` and raises an `ExBadAssert` exception on failure. The (optional) parameters *str, func, file* and *line* are transfered to `THROW` for debugging purpose. The `test_invariant` assertion relies on the message `ginvariant` which must be specialized for `MilliCounter` to be effective:

```
defmethod(OBJ, ginvariant, MilliCounter,
                           (STR)func, (STR)file, (int)line)
  next_method(self); // check Counter invariant first
  test_assert(self->mcnt >= 0 && self->mcnt < 1000,
       "out of range", func, file, line); // trace location
endmethod
```

Contracts and inheritance Bertrand Meyer recommends to evaluate inherited contracts as a *disjunction* of the preconditions and as a *conjunction* of the postconditions, but [26] demonstrates that EIFFEL-style contracts may introduce behavioral inconsistencies. Thus COS prefers to treat both pre and post conditions as conjunctions since this is the only known solution compatible with multi-methods where subtyping is superseded by the *class precedence list*.

Contracts levels The level of contracts can be set by defining the macro `COS_CONTRACT` to one of the following cumulative levels: `NO` disable contracts, `COS_CONTRACT_PRE` enables `PRE` sections (default), `COS_CONTRACT_POST` enables `POST` sections, `COS_CONTRACT_ALL` enables invariant (debug).

Exceptions The exception mechanism of COS is based on the pair `setjmp`/`longjmp` and provides full-fledged `TRY-CATCH-FINALLY` statements with the same semantic as in OO languages. `THROW` (resp. `CATCH`) relies on message `gthrow` (resp. `gisKindOf`) to throw (resp. identify) the exception.

4.4 Properties

Property declaration is a useful programming concept which allows, amongst others, to manage the access of object attributes, to use objects as associative arrays or to make objects persistent. Properties in Cos are just syntactic sugar on top of the definition of class-objects and the specialization of the accessors `ggetAt` and `gputAt` already mentioned.

Definition Properties are class-objects deriving from the class `Property` (fig. 3) with lowercase names prefixed by `P_`.

```
defproperty( name );
defproperty( size );
defproperty( class );
defproperty( value );
```

For example, the `value` property definition is equivalent to:

```
defclass(P_value, Property) endclass
```

Class properties Once properties have been defined, it is possible to define class-properties:

```
defproperty(Counter, (cnt)value, int2OBJ, gint);
defproperty(Counter, ()class, gclass); // read−only
```

with:

```
OBJ int2OBJ(int val) { // no receiver: cannot be a method
  return gautoDelete( aInt(val) ); // boxing
}
```

The `value` property is associated with the `cnt` attribute with read-write semantic and uses user-defined boxing (`int2OBJ`) and unboxing (`gint`). The `class` property is associated with the entire object (omitted attribute) with read-only semantic and uses the inherited method `gclass` to retrieve it.

Sometimes the abstraction or the complexity of the properties require handwritten methods. For instance:

```
defmethod(OBJ, ggetAt, Person, mP_name)
  retmethod( gcat(self->fstname, self->lstname) );
endmethod
```

is, assuming `gname(Person)` is doing the `gcat`, equivalent to:

```
defproperty(Person, ()name, gname);
```

Using properties The code sample below prints some object property (or raise the exception `ExBadMessage`):

```
void print_property(OBJ obj, OBJ prp) {
  gprint( ggetAt(obj, prp) );
}
```

5. PERFORMANCE

In order to evaluate the efficiency of Cos, small test suites[11] have been written to stress the message dispatcher in various conditions. The test results summarized in table 1 have been performed on an Intel DualCore2™ T9300 Cpu 2.5 Ghz with Linux Ubuntu 64-bit and the compiler Gcc 4.3 to compile the tests written in the three languages. The timings have been measured with `clock()` and averaged over many loops of $2 \cdot 10^8$ iterations each. The *Param.* column indicates the number of parameters of the message split by *selectors* (open types) and *arguments* (closed types). The other columns represent the performances in million of invocations sustained per second for respectively C++ virtual member functions, Objective-C messages and Cos messages. The tests stress the dispatcher with messages already described in this paper: `incr` increments a counter, `incrBy{2..5}`$_{\text{opt}}$ accept from 1 to 5 extra *closed* parameters (to stress the construction of `_arg`) and `addTo{2..4}`$_{\text{opt}}$ add from 2 to 5 `Counter`s together (to stress multiple dispatch). Multiple dispatch has been implemented with the Visitor pattern in C++ and Objective-C.

Tests	Param.	C++	ObjC	Cos
single dispatch				
`counter incr`	$1+0$	176	122	218
`counter incrBy`	$1+1$	176	117	211
`counter incrBy2`	$1+2$	176	115	185
`counter incrBy3`	$1+3$	176	112	171
`counter incrBy4`	$1+4$	167	111	154
`counter incrBy5`	$1+5$	167	107	133
multiple dispatch				
`counter addTo`	$2+0$	90	40	150
`counter addTo2`	$3+0$	66	23	121
`counter addTo3`	$4+0$	45	16	90
`counter addTo4`	$5+0$	40	12	77

Table 1: Performances in 10^6 calls/second

Concerning the performance of *single dispatch*, Cos shows a good efficiency since it runs in average at about the same speed as C++ and about ×1.6 faster than Objective-C. On one hand, Cos efficiency decreases faster than C++ because it passes more hidden arguments (*i.e.* `_sel` and `_arg`) and uses more registers to compute the dispatch. On the other hand, C++ shows some difficulties to manage efficiently multiple inheritance of abstract classes. Concerning the performance of *multiple dispatch*, Cos outperforms both C++ and Objective-C by factors ×1.9 and ×5.3 respectively. Concerning the performance of *message forwarding*, we have seen that by design, it runs at the full speed of message dispatch in Cos. Rough measurements of Objective-C message forwarding (linear lookup) shows that Cos performs from ×50 to ×100 faster, depending on the classes.

Multi-threading The same performance tests have been run with Posix multi-threads enabled. When the Thread-Local-Storage mechanism is available (Linux), no significant impact on performance has been observed (<1%). When the architecture supports only Posix Thread-Specific-Key (Mac OS X), the performance is lowered by a factor ×1.6 and becomes clearly the bottleneck of the dispatcher.

Object creation Like other languages with semantic by reference, Cos loads heavily the C memory allocator (*e.g.* `malloc`) which is not very fast. If the allocator is identified as the bottleneck, it can be replaced with optimized pools by overriding `galloc` or by faster external allocators (*e.g.* Google `tcmalloc`). Cos also takes care of automatic objects which can be used to speed up the creation of local objects.

Other aspects Other features of Cos do not involve such heavy machinery as in message dispatch or object creation. Thereby, they all run at full speed of C: *contracts* run at the speed of the user tests since the execution path is known at compile time and flattened by the optimizer, empty `TRY`-*blocks* run at the speed of `setjmp` which is a well known bottleneck and `next_method` runs at 70% of the speed of an indirect function call (*i.e.* late binding) because it also has to

[11] The code can be found in the module `CosBase/tests`.

pack the closed arguments into the `_arg` structure. Finally, this kind of benchmark is a worst-case (resp. best-case) for Cos (resp. C++) since in real applications, its *collaborative* features increase (resp. decrease) its *relative* performance.

6. GENERIC DESIGN *(COMPONENTS)*

This overview of Cos shows that the principles stated in the introduction are already well fulfilled. So far:

Simplicity can be assumed from the fact that a large part of Cos syntax can be described within few pages, including some examples, implementation details and comparisons with other languages. Besides, the language has a simple and consistent syntax and semantics free of pitfall.

Flexibility and extensibility come from the nature of the object model which allows extending (methods bound to generics), wrapping (around methods) or renaming (method aliases) *behaviors* with a user-friendly syntax. Moreover encapsulation, polymorphism, messages (loose coupling) and contracts are also strong assets for software extensibility.

Reusability and variability come from the key concepts of Cos which enhance generic design: polymorphism, collaboration (multi-methods) and composition (delegation).

Efficiency measurement shows that its runtime features perform well compared to other mainstream languages.

Portability comes from its nature: a C89 library.

It is widely acknowledged that dynamic programming languages simplify significantly the implementation of classical design patterns [22] when they don't supersede them by more powerful dynamic patterns [6, 27, 28]. This section focuses on how to use Cos features to simplify design patterns or to turn them into reusable components, where the definition of *componentization* is borrowed from [11, 12]:

Encapsulation+Polymorphism+Multi-dispatch+Delegation

6.1 Simple Patterns

Creational Patterns It is a well known that these patterns vanish in languages supporting generic types and introspection. We have already seen `gnew`, here is more:

```
OBJ gnewWithStr(OBJ cls, STR str) {
  return ginitWithStr(galloc(cls), str);
}

OBJ gclone(OBJ obj) {
  return ginitWith(galloc(gclass(obj)), obj);
}
```

The Builder pattern is a nice application of property metaclasses to turn it into a so-called Class Cluster:

```
defmethod(OBJ, galloc, pmString)
  retmethod(_1); // lazy, delegate the task to initializers
endmethod

defmethod(OBJ, ginitWithStr, pmString, (STR)str)
  OBJ lit_str = galloc(StringLiteral);
  retmethod( ginitWithStr(lit_str, str) );
endmethod
```

This example shows how to delegate the object build to the initializer which in turn allocates the appropriate object according to its arguments, an impossible task for the allocator. The allocation of the `StringLiteral` uses the standard allocator inherited from `Object`, despite it derives from the class `String`, thanks to property metaclass. Now, the code:

```
OBJ str = gnewWithStr(String,"literal string");
```

will silently return an instance of `StringLiteral`. This is the principle of Class Clusters where the front class (*e.g.* `String`) delegates to private subclasses (*e.g.* `StringLiteral`) the responsibility to build the object. It is worth noting that each `pmString` specialization needed to handle new subclasses is provided by the subclass itself, thanks to the open class model, which makes the Builder pattern truly extensible. Most complex or multi-purpose classes of Cos are designed as class clusters (*e.g.* `Array`, `String`, `Functor`, `Stream`).

Garbage Collector This exercise will show how to simplify memory management in Cos with only few lines of code. We start by wrapping the default object allocator such that it *always* auto-releases the allocated objects:

```
defmethod(OBJ, (galloc), mObject) // around method
  next_method(self); // allocate
  gautoRelease(RETVAL); // auto−release
endmethod
```

Then we neutralize (auto-)delete and reinforce retain:

```
defmethod(void, (gdelete), Object) // do nothing
endmethod

defmethod(OBJ, (gautoDelete), Object) // clone auto objects
  BOOL is_auto = self->rc == COS_RC_AUTO;
  retmethod( is_auto ? gclone(_1) : _1 );
endmethod

defmethod(OBJ, (gretain), Object)
  next_method(self); // retain
  if (self->rc == COS_RC_AUTO)
    RETVAL = gretain(RETVAL); // once more for auto objects
endmethod
```

Now, the following code:

```
OBJ pool = gnew(AutoRelease);
for(int i = 0; i < 1000; i++)
  OBJ obj = gnewWithStr(String, "string");
gdelete(pool); // pool specialization, collect the strings
```

does not create any memory leak, there is no longer the need to delete or auto-delete your objects. For the first runs, the default auto-release pool managed by Cos can be used. Then a memory profiler will show the appropriate locations where intermediate auto-release pools should be added to trigger collects and limit the memory usage.

Key-Value-Coding We have already seen that properties allow accessing object attributes, but to implement KVC, we need to translate strings (key) into properties (noun):

```
defmethod(OBJ, ggetAt, Object, String)
  OBJ prp = cos_property_getWithStr(self2->str);
  if (!prp) THROW(gnewWith(ExBadProperty,_2));
  retmethod( ggetAt(_1, prp) );
endmethod

defmethod(void, gputAt, Object, String, Object)
  OBJ prp = cos_property_getWithStr(self2->str);
  if (!prp) THROW(gnewWith(ExBadProperty,_2));
  gputAt(_1, prp, _3);
endmethod
```

where `cos_property_getWithStr` is an optimized version of `cos_class_getWithStr` for properties from the API of Cos, which also provides `cos_class_{read,write}Properties` to retrieve all the properties of a class (and its superclasses).

Key-Value-Observing Adding access notifications of properties is the next step after KVC, using around methods:

```
defmethod(OBJ, (ggetAt), Person, mP_name)
  useclass(After, Before); // local declaration
  OBJ context = aMthCall(_mth,_1,_2,_arg,_ret);// this call
  gnotify(center, context, Before);
  next_method(self1, self2); // get property
  gnotify(center, context, After);
endmethod
```

where `_mth` is the object representing the method itself. This example assumes that observers and objects observed have been registered to some notification `center` as commonly done in the Key-Value-Observing pattern.

6.2 Proxies and Decorators

Proxy Almost all proxies in Cos derive from the class `Proxy` which handles some aspects of this kind of class:

```
defclass(Proxy);
  OBJ obj; // delegate
endclass

defmethod(void, gunrecognizedMessage2, Proxy, Object)
  forward_message(self1->obj, _2);
  check_ret(_sel, _ret, self1);
endmethod

defmethod(void, gunrecognizedMessage2, Object, Proxy)
  forward_message(_1, self2->obj);
  check_ret(_sel, _ret, self2);
endmethod

// ... other rank specializations
```

where the small function `check_ret` takes care to return the proxy when the forwarded message returns the delegate *obj*.

Tracer For the purpose of debugging, Cos provides the simple proxy `Tracer` to trace the life of an object:

```
defclass(Tracer,Proxy) // usage: gnewWith(Tracer, obj);
endclass

defmethod(void, gunrecognizedMessage2, Tracer, Object)
  trace_msg2(_sel, self1->Proxy.obj, _2);
  next_method(self1, self2); // forward message
endmethod

defmethod(void, gunrecognizedMessage2, Object, Tracer)
  trace_msg2(_sel, _1, self2->Proxy.obj);
  next_method(self1, self2); // forward message
endmethod

// ... other rank specializations
```

where `trace_msg2` prints useful information on the console.

Locker The locker is a proxy which *avoids synchronization deadlock* on shared objects that can be encountered in programming languages supporting only single-dispatch [7]:

```
defclass(Locker,Proxy) // usage: gnewWith(Locker, obj);
  pthread_mutex_t mutex;
endclass

defmethod(void, gunrecognizedMessage2, Locker, Object)
  lock(self1); // lock the mutex
  next_method(self1,self2); // forward the message
  unlock(self1); // unlock the mutex
endmethod

defmethod(void, gunrecognizedMessage2, Locker, Locker)
  sorted_lock2(self1,self2); // lock by sorted addresses
  next_method(self1,self2); // forward the message
  sorted_unlock2(self1,self2); // unlock by sorted addresses
endmethod

// ... other 55 specializations
```

For the sake of efficiency, higher ranks use *sorting networks*.

Multiple Inheritance The first version of Cos was natively implementing multiple inheritance using the C3 algorithm [29] to compute the *class precedence list* on the way of Dylan, Python and Perl6. But it was quickly considered as too complex for the end-user and incidental as far as fast generic delegation could be achieved. Indeed, multiple inheritance can be simulated by *composition* and *delegation* with an efficiency close to native support[12] as shown below:

```
defclass(IOStream, OutStream) // inherits its out_stream
  OBJ in_stream;
endclass

defmethod(void, gunrecognizedMessage1, IOStream)
  forward_message(self->in_stream);
endmethod
```

Now, messages of rank one not understood by the `IOStream`s (*e.g.* `gget`, `gread`) will be forwarded to their `InStream`. Albeit it is more efficient to use *dynamic inheritance* to switch `IOStream`s back and forth between `OutStream` and `InStream`.

Distributed Objects Without going into the details, we can mention that Cos already implements all the key concepts required to develop a distributed object system on the model of Objective-C and Cocoa. A challenge for the future.

6.3 Closures and Expressions

Cos provides the family of `geval`*n* messages (equivalent to Common Lisp `funcall`) and the class cluster `Functor` to support the mechanism of closures and more generally lazy expressions and higher order messages. The objects representing the context of the closure (*i.e.* the free variables) are passed to the `Functor` constructor which handles partial evaluation and build expressions. The example hereafter shows another way to create a counter in Perl using a closure:

```
1  sub counter {
2    my($val) = shift; # seed
3    $cnt = sub { # closure
4      my($inc) = shift; # increment
5      return $val += $inc; # perform addTo
6    };
7    return $cnt; # return the closure
8  }
9
10 $cnt = counter(0);
11 for($i=0; $i<25000000; $i++) {
12   &$cnt(2);
13 }
```

which can be translated into Cos as:

```
1  OBJ counter(int seed) {
2    return gaddTo(aCounter(seed), aVar(0));
3  }
4
5  int main(void) {
6    OBJ cnt = counter(0);
7    for(int i=0; i<25000000; i++)
8      geval1(cnt, aInt(2));
9  }
```

[12] Objective-C delegation is far too slow to simulate MI.

Line 2 creates a closure using the placeholder `aVar(0)` which triggers the building of the functor (lazy expression) and *deduces* its arity (here 1) from the remaining parameters, namely the seed *boxed* in the counter. As one can see, COS achieves the same task as PERL with about the same amount of code but runs more than $\times 16$ *faster*.

The following example shows a more advanced example involving lazy expressions and indexed placeholders:

```
// return f' = (f(x + dx) − f(x))/dx
OBJ ggradient(OBJ f) {
  OBJ x = aVar(0);                          // placeholder #1
  OBJ dx = aVar(1);                         // placeholder #2
  OBJ f_x = geval1(f, x);                   // lazy expression
  OBJ f_xpdx = geval1(f, gadd(x, dx));      // lazy expression
  return gdiv(gsub(f_xpdx, f_x), dx);       // lazy expression
}

// return f'(x)|dx
OBJ gderivative(OBJ f, OBJ dx) {
  return geval2(ggradient(f),aVar(0),dx);// lazy expression
}
```

Now, we can map this function to a `bag` of values (strict evaluation) or expressions (lazy evaluation) indifferently:

```
OBJ df = gderivative(f, dx);
OBJ new_bag = gmap(df, bag);
```

Higher Order Messages The principle behind HOMS is to incrementally construct on-the-fly expressions from their arguments and evaluate the last expression built to get the result (*i.e.* interpreter). OBJECTIVE-C HOMS [31] are not generic and require conventions and user-defined methods (per class) to satisfy the underlying machinery. C++ Higher Order Metaprogramming [32] relies on traits and templates to build static meta-expressions but still requires significant user-defined code. Finally, HOMS in COS rely on polymorphism, delegation and multi-methods to build efficient HOMS as *generic components*: a nice proof of the power of COS. Comparing to [31], COS fast generic delegation *avoids the need* to cache messages in HOM objects while multi-methods *avoid the need* to provide specific HOM objects per task (*e.g.* `filter`, `select`, `collect`) and methods per class (*e.g.* `Array`, `List`, `Map`) and to build heavy machinery (Visitor pattern) behind the scene to handle their collaboration.

7. CONCLUSION

COS seems to be unique by the set of features it provides. The library approach on top of the C programming language without requiring any extra preprocessor, compiler or platform specific feature, allowed to explore rapidly some object models and to select the most appropriate one fulfilling the seven aimed principles: *simplicity, flexibility, extensibility, reusability, variability, efficiency* and *portability*. Moreover, the list of COS features is complete and consistent: *syntax to support object-oriented programming, uniform object model with extended metaclass hierarchy, multi-methods, fast generic delegation, design by contract, properties and key-value coding, exceptions, ownership and closures*. COS features have been optimized from the design point of view, but for the sake of simplicity and portability, code tuning was never performed and lets some room for future improvement. The 8000 lines of source code of COS can be downloaded from sourceforge.net under the LGPL license.

8. REFERENCES

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

[2] D.R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. 1997.

[3] J.K. Foderaro. *Lisp is a Chameleon*. 1991.

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

[5] *The Objective-C 2.0 Programming Language*. 2008.

[6] *Cocoa Fundamentals Guide*. Apple Inc., 2007.

[7] S.E. Keene. *Object Oriented Programming in Common Lisp: A Programmers Guide to* CLOS. 1989.

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

[9] J. Bosch. *Design of an Object-Oriented Framework for Measurement Systems*. John Wiley & Sons, 2000.

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

[11] K. Rege. *Design Patterns for Component-Oriented Software Development*. Euromicro'99, vol. 2, 1999.

[12] B. Meyer, K. Arnout. *Pattern Componentization: The Visitor Example*. Computer, vol. 39, no. 7, July 2006.

[13] B. Meyer, K. Arnout. *Pattern Componentization: The Factory Example*. Software Engineering, July 2006.

[14] R.E. Johnson. *Dynamic Object Model*. 1998

[15] D. Riehle, M. Tilman and R.E. Johnson. *Dynamic Object Model*. PLoP'2000.

[16] J.W. Yoder and R.E. Johnson. *The Adaptive Object-Model Architectural Style*. WICSA'2002.

[17] J. van Gurp, and J. Bosch. *D&E of Object-Oriented Frameworks: Concepts & Guidelines*. March 2001.

[18] D. Parsons and al. *An architectural pattern for component-based application frameworks*. 2005.

[19] H. Boehm. *Bounding Space Usage of Conservative Garbage Collectors*. PoPL'2002.

[20] R. Razavi and al. *Language support for Adaptive Object-Models using Metaclasses*. ESUG'2004.

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

[22] E. Gamma and al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

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

[24] P. Pirkelbauer, Y. Solodkyy and B. Stroustrup. *Open Multi-Methods for C++*. GPCE'07.

[25] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[26] R.B. Findler and al. *Behavioral Contracts and Behavioral Subtyping*. FSE'2001.

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

[28] G.T. Sullivan *Advanced Programming Language Features for Executable Design Pattern*. MIT 2002.

[29] K. Barrett and al. *A monotonic superclass linearization for Dylan*. OOPSLA'96.

[30] *The Cecil Standard Library*. 2004.

[31] M. Weiher and S. Ducasse. *Higher-Order Messaging*. OOPSLA'05.

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