

Objective Modula-2 – A Discussion and Working Paper

First release, revision 4 - August 2005, work in progress

(c) Sunrise Telephone Systems Ltd., Author BK

Introduction

The notion to create Objective Modula-2 was born out of the wish for an alternative to C syntax as the base for development using the Cocoa or GNUstep frameworks while at the same time preserving the Smalltalk derived method declaration and message passing syntax of Objective-C. In other words, the aim is to create a "better" Objective-C in which the C syntax is replaced by Modula-2 syntax and the Smalltalk derived syntax remains, hence the name "Objective Modula-2". As a result, Modula-2 purists may find some of the syntax discussed in this paper far too much unlike Modula-2. The intent is to strike a balance. As a general guideline, the transformation of C to Objective-C shall be used as a blueprint how to derive Objective Modula-2 from Modula-2. In other words, Modula-2 is to receive Smalltalk derived add-ons in the very same manner C did for its evolution into Objective-C.

Objective-C's message passing syntax is based on Smalltalk which uses an interleaved infix-like notation for parameters. For example, one of the factory methods of the Cocoa class NSString is called 'initWithData:encoding:error:' and its parameters 'data', 'enc' and 'err' are interleaved with components in the method's name both in the declaration and invocation of the method. The method's name is called a signature in Objective-C terminology. The method signature is not to be confused for a function call with embedded named parameters and the colons are not to be confused for separator symbols. All characters including semicolons, from the first alpha character to the last semicolon character, are an integral part of the method's identifier.

From the viewpoint of a Modula-2 programmer this syntax is very unusual, just as it is to C programmers. Yet, Objective-C and Cocoa/GNUstep owe a great deal of their clarity and expressiveness to this syntax.

Most if not all approaches to integrating languages with Cocoa/GNUstep have chosen to use a more conventional syntax such as is common in Ada/C/Java/Modula-2/Pascal but this comes at the expense of sacrificing the clarity and expressiveness and also the simplicity found in Objective-C. For example, the Python-ObjC bridge translates the colons in the method's signature into underscore characters "_" and their corresponding parameters are then provided in form of a comma separated list enclosed in parentheses following the method's signature to make it look like a function call. The method 'initWithData:encoding:error:' becomes 'initWithData_encoding_error_(data, enc, err)'. This leads to several disadvantages, a discussion of which is out of scope of this document.

Objective-C programmers, most of whom were former C programmers, unanimously state that they found the Smalltalk derived syntax extremely weird at the beginning but quickly learned to appreciate it and that they would not want to miss it. The author can only second this experience. Ada, Modula-2 and Pascal programmers are likely to adjust as easily and learn to appreciate the benefits of this notation, most notably clarity and expressiveness, which means better readability and maintainability of code. It also has the side effect to make developers think in terms of messages, rather than function calls when invoking methods.

At the time of writing the first release of this paper, Apple Computer Inc. had just announced that they would introduce Intel x86 based Macintosh computers within a year and they released a new version of their Xcode IDE which includes tools to migrate PPC code to so called universal binary code with support for both PPC and x86. The migration of Cocoa based applications appears to be very straightforward. In many cases all that is needed is checking a compiler option and recompile. However, migration of Carbon based applications will not be as straightforward. Worst off are developers who have used Code Warrior instead of Xcode. The only migration path for them is to first migrate their applications to Xcode and then migrate towards universal binaries. Users of the p1 Modula-2 compiler will probably find themselves in a very similar situation. Apple's Carbon framework mainly intended for support of pre-MacOS X legacy code is the procedural counterpart of the object oriented Cocoa framework, which is an evolution of OpenStep. All Pascal

and Modula-2 programmers on the Macintosh have been using Carbon APIs instead of Cocoa because there has not been any way for them to utilise Cocoa.

With Apple's Intel based Macintosh computers in sight and the pain of migrating Carbon based applications, many more developers will become interested in Cocoa. Yet the prospect of having to migrate to Objective-C, a superset of C, may spoil it for those who prefer to use Pascal and Modula-2. To tell the truth, Objective-C is far more pleasant than C because most of the code is written in the added-on Smalltalk derived message passing syntax. However, if there was a variant such as the Objective Modula-2 language this paper proposes, then it would very likely emerge as the favourite migration path to Cocoa for many developers who have been sitting on the fence because they didn't like the "C" in Objective-C.

Likewise, Apple's embrace of Intel x86 is likely to result in an increased interest in GNUstep on both Linux and Windows. Here again, having the choice between a C style and a Pascal/Modula-2 style based syntax for development is likely to lower the entry barrier for those developers who don't like C.

In turn adoption of Objective Modula-2 for Cocoa and GNUstep development may well renew interest in Modula-2 itself.

Implementation Requirements (Modula-2 Standards)

The author of this paper is not very fond of the ISO Modula-2 standard, even though he was one of the members of the ISO standardisation committee (aka WG13) in the late 1980s. In any event, for the purpose of ObjM2 and Cocoa/GNUstep development the ISO Modula-2 standard is largely irrelevant.

For one, this applies to the standardisation of libraries. Since the purpose of ObjM2 is Cocoa and GNUstep development, traditional Modula-2 libraries are irrelevant with the exception of SYSTEM and ASCII. In most cases, an ObjM2 programmer will be using the Cocoa/GNUstep and Core Foundation frameworks instead of traditional Modula-2 libraries, just like an Objective-C programmer mostly uses those frameworks instead of C libraries. In some cases, outside of the Cocoa/GNUstep and Core Foundation frameworks, the Modula-2 libraries M2RTS, IO, Mathlib, Storage and the C library libc may also be useful, but it is very unlikely that any others would be used.

The same applies to the object oriented extensions to ISO Modula-2. Since the raison d'etre of ObjM2 will be to natively interface to Cocoa/GNUstep and its underlying Objective-C based object model, Smalltalk derived OO extensions equivalent to those of Objective-C will replace the OO extensions in ISO Modula-2.

Consequently, in order for ObjM2 to be implemented and ready to allow Cocoa/GNUstep development, PIM or base ISO standard support without any library support other than SYSTEM will be sufficient. A foreign library interface to libc may however be useful, if not required.

Design Goals

Create a new language, here called "Objective Modula-2", or in short "ObjM2" which is derived from Modula-2 by adding Smalltalk like method declaration and message passing syntax in the very same way Objective-C was derived from C.

Maintain Modula-2's original philosophy of simplicity.

Allow the new language to use Cocoa/GNUstep as well as custom libraries and frameworks written in Objective-C. Allow the new language to be used to create Cocoa/GNUstep as well as custom libraries and frameworks which can be used within Objective-C code as if it was written in Objective-C.

Where possible without violation of the simplicity philosophy, make the language attractive to users of other languages, in particular to C and Objective-C programmers.

Implementation Goals

As a first step, create a preprocessor which will translate ObjM2 into ObjC and release it under a suitable open source license such as a BSD style license or the GNU General Public License Version 2.

Integrate the preprocessor with GCC and Apple's Xcode IDE so that the use of ObjM2 becomes seamless when developing in Xcode. Possibly integrate into GNUstep's ProjectBuilder equivalent, too, subject to somebody from the GNUstep community joining in.

Possibly integrate the preprocessor into the GNU Modula-2 front end for GCC.

NB: The first implementations of Objective-C compilers were implemented as preprocessors to then available C compilers.

Syntax

1) C++ style comments

In addition to M2 multiline comments (* *) ObjM2 inherits EOL terminated comments from C++

Example:

```
// this is a comment
```

2) C style increment and decrement operators

ObjM2 inherits the built-in increment and decrement postfix operators from C.

Examples:

```
counter++; // increment counter  
counter--; // decrement counter
```

3) C derived ternary operator in Modula-2 style syntax

ObjM2 inherits the concept behind the ternary operator of C, but Modula-2 style syntax shall be used instead of C syntax.

Example:

```
// if cond is true the expression on the right evaluates to t_value  
// otherwise it evaluates to f_value  
// cond is boolean; value is arbitrary;  
// t_value and f_value must evaluate to the same type as value  
value := TEVAL(cond, t_value, f_value);
```

NB: this should probably be a built-in as it is intended to compile into inline code but it may well be put into a module such as SYSTEM, from where it would then have to be imported before it can be used. A keyword other than "TEVAL" may be used eventually.

4) Objective-C style method declarations

ObjM2 may need to be able to understand Objective-C style method declarations in order to read C header files of ObjC class modules. This raises the question whether it should retain the ObjC method declaration syntax at least as an available option, perhaps with Modula-2 style type designators.

Examples:

```
// declare instance method isEmptyString
- (BOOL)isEmptyString;

// declare class method stringWithString
+ (POINTER TO NSString)stringWithString:(POINTER TO NSString)aString;
```

While the closing semicolon of a method declaration is optional in an Objective-C implementation module, the closing semicolon shall always be mandatory in ObjM2.

5) M2 style method declarations

In addition to the ability to understand foreign Objective-C method declarations, ObjM2 shall also provide its own M2 derived syntax.

Examples:

```
// declare instance method isEmptyString
INSTANCE METHOD isEmptyString : BOOL;

// declare class method initWithData:encoding:error:
CLASS METHOD initWithData: (data : POINTER TO NSData)
                    encoding: (enc : NSStringEncoding)
                    error: (VAR POINTER TO NSError) : NSObject;
```

While the closing semicolon of a method declaration is optional in an Objective-C implementation module, the closing semicolon shall always be mandatory in ObjM2.

6) Objective-C style message passing

ObjM2 retains Objective-C's message passing syntax as is.

Examples:

```
// sending the message isEmpty to myString of class NSString
IF [myString isEmpty] THEN ...

// showing nested message passing
IF [[path stringByStandardizingPath] isAbsolutePath] THEN ...

// initialising myString sending the message stringWithData:encoding:error:
// to the NSString class object
myString := [NSString stringWithData:myData
                    encoding:NSUTF8StringEncoding
                    error:ADDR(err)];
```

NB: in the above example err is passed as ADDR(err) because it is the equivalent of C's &err notation. However, if we use the (VAR POINTER TO ...) syntax, then we may as well let the compiler insert the address of err there for us and use the simpler error:err syntax, although library compatibility issues may arise. This will require some further investigation.

7) Interface and implementation module headers

ObjM2 shall use Modula-2 style syntax for interface and implementation module headers.

Objective-C's "@interface" shall become "INTERFACE MODULE" **for class modules** in ObjM2, "@implementation" becomes "IMPLEMENTATION MODULE" and "@end" becomes "END". For conventional Modula-2 library modules, the "DEFINITION MODULE" syntax shall remain.

a) implementing a class

In Objective-C the class name is followed by a colon and the name of it's superclass. The same shall apply to ObjM2 definition modules defining and implementation modules implementing a class.

Examples:

```
INTERFACE MODULE MyClass : NSObject;  
// instance variables  
// method declarations  
END MyClass.
```

```
IMPLEMENTATION MODULE MyClass;  
// method implementations  
END MyClass.
```

Objective-C declares instance variables of a class in the interface module and uses the directives "@private", "@protected" and "@public" to define their visibility and scope. ObjM2 shall introduce Modula-2 style keywords or directives in place of @private, @protected and @public and also introduce an additional access mode. The four access modes shall be as follows ...

a) public

the variable is visible everywhere without any restrictions

b) protected

the variable is visible within every subclass

c) semi-private

the variable is visible to every class declared within the same library module, to facilitate sharing of variables between classes in a class cluster but without opening the variable to user defined subclasses.

d) private

the variable is visible only within the class it belongs to

Examples:

```
INTERFACE MODULE MyClass : NSObject;  
  
INSTANCE VAR  
  REVEAL      (* seen as @public when used from ObjC *)  
  pub : Pub;  
  PROTECTED   (* seen as @protected when used from ObjC *)  
  foo : Foo;  
  RESTRICTED  (* seen as @private when used from ObjC *)  
  bar : Bar;  
  PRIVATE     (* seen as @private when used from ObjC *)  
  taboo: Boo;
```

An instance variable defined with RESTRICTED access mode is visible throughout the library module in which the class module is declared, but not beyond. This is analogous to Java's package level access mode. The default access mode (in the event that nothing is specified) shall be PRIVATE. The names of keywords may still be revised.

b) implementing a category

Objective-C allows to add further methods to an existing class outside of the modules that define and implement that class. This is called a category in Objective-C terminology. In order to define and implement a category, an arbitrary name for the category enclosed in parentheses follows the class name in both the interface and the implementation module, for example ...

```
// Category MyAdditionsToNSString of class NSString
@interface NSString (MyAdditionsToNSString)
```

ObjM2 shall use a more Modula-2 like syntax introducing a new keyword "EXTENDS" to follow the category name itself followed by the class name. A semicolon following the class name shall be mandatory (unless the class adopts one or more protocols - see "adopting a formal protocol").

Example:

```
INTERFACE MODULE MyAdditionsToNSString EXTENDS NSString;
```

c) declaring a formal protocol

Objective-C allows to define interfaces for methods unattached to any class but which any given class may implement. This makes the method declarations independent of the class hierarchy which is useful for the implementation of delegates. There are formal and informal protocols. Informal protocols are declared using categories. Formal protocols require additional syntax ...

```
@protocol myProtocol;
// method declarations
@end
```

ObjM2 shall use a more Modula-2 like syntax introducing a new keyword "PROTOCOL" to precede the keyword "INTERFACE" in the interface module header.

Example:

```
PROTOCOL INTERFACE MODULE MyProtocol;
```

d) adopting a formal protocol

In Objective-C a class is said to "adopt" a formal protocol if it agrees to implement the methods the protocol declares. Class declarations list the names of adopted protocols within angle brackets after the superclass name ...

```
@interface ClassName : Superclass < MyProtocol, someOtherProtocol >
```

ObjM2 shall use a more Modula-2 like syntax introducing a new keyword "ADOPTING" to follow the superclass name in a module header itself followed by a comma separated list of protocol names terminated by a semicolon.

Examples:

```
INTERFACE MODULE MyClass : NSObject ADOPTS MyProtocol;
```

```
INTERFACE MODULE MyAdditionsToNSString EXTENDS NSString ADOPTS MyProtocol;
```

8) Exceptions

The keywords `@throw`, `@try`, `@catch`, `@finally` in Objective-C shall be replaced by Modula-2 style keywords "RAISE", "TRY", "ON ... DO" and "CONTINUE" in order to avoid name clashes with ISO Modula-2's exceptions.

Example:

```
// raising an exception
RAISE(anException);

// alternatively, an exception can also be raised by sending the
// raise message to an object of class NSEException ...
[anException raise];

// catching an exception
TRY
// statements
ON anException DO
// statements handling the exception
CONTINUE
// statements always executed after leaving try and catch blocks
END;
```

NB: in Objective-C, exceptions are understood in the literal meaning of the word "exception". They are intended for use in exceptional circumstances only. They are not meant to be used for ordinary error handling. For example, Cocoa raises an exception when an argument is out of range, such as when a search in a string is specified to be performed beyond the length of the string. For ordinary error handling, such as errors that may occur when trying to read or write a file, an error parameter passed by reference is used which points to an object of class NSError and its methods are then used to deal with error handling and reporting.

9) Thread synchronisation

The keyword `@synchronized` in Objective-C shall be replaced by a Modula-2 style keyword "SYNCHRONIZED". Synchronization in ObjM2 shall support recursive and reentrant code, just as it does in Objective-C.

Example:

```
// locking a method using a semaphore
SYNCHRONIZED(aSemaphore)
// critical code
END;
```

10) Introspection

The keywords `@selector` and `@protocol` in Objective-C shall be replaced by Modula-2 style keywords "SELECTOR" and "PROTOCOL".

Examples:

```
// using the SELECTOR directive
IF [[self class] respondsToSelector:SELECTOR(isEmpty)] THEN ...
```

```
// using the PROTOCOL directive
IF [[self class] conformsToProtocol:PROTOCOL(myProtocol)] THEN ...
```

STILL TO DO:

11) Distributed objects

oneway, in, out, inout, bycopy, byref

12) misc

The Keyword `@defs` in Objective-C may either be replaced by a Modula-2 style keyword "DECLARATION" or any other suitable keyword. Alternatively, this feature may not be implemented at all because it is rather unsafe.

Example:

```
// using the DECLARATION directive
myDef RECORD
    DECLARATION(someClass) // reproduces the declaration of someClass
END;
```

others to do ...

id, Class, BOOL, SEL, IMP, nil, Nil, @class, @encode()

13) EBNF

14) Objective-C reference

More information on the Objective-C language can be found at

http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/chapter_1_section_1.html#apple_ref/doc/uid/20001422

[end of this document]