Concurrent Object Oriented 'C' (COOC)

Rajiv Trehan、澤島信介、森下明、友田一郎、井上淳、前田賢一

(株) 東芝 総合研究所 情報システム研究所

並行オブジェクト指向言語COOCは、オブジェクトがデータ遮蔽の境界のみでなく、プロセスの境界ともなり得るものであり、それぞれのオブジェクトが並行に動くという前提に基づく言語である。COOC言語モデルにおいては wait-when-neccessary 型の通信を採用し、共有資源に対する保護も(排他的メソッド実行という形で)提供している。またCOOC実行時システムはオブジェクトとそのメソッド実行について、いくつかの異なるパラダイムを提供している。

COOCはC言語の拡張であり、C, C++およびObjective-Cと上位互換である。COOCの基本的な文法はObjective-Cと同じであり、COOCのコードはコンパイラによってCに変換される。

本論文ではCOOCの並行オブジェクトモデルおよび実行時システム、種々の実行ポリシーを支援する機構について説明する。

Concurrent Object Oriented 'C' (COOC)

Rajiv Trehan, Nobuyuki Sawashima, Akira Morishita, Ichiro Tomoda,
Atsushi Inoue, Ken-ichi Maeda

Information Systems Laboratory, Research and Development Center, Toshiba Corp.

Concurrent Object Oriented C (COOC) is a language based on the premise that an object not only provides an encapsulation boundary but can also form a process boundary; each object acts concurrently. The COOC language model employs a wait-when-necessary communication and affords shared resource protection (via exclusive method invocation). The COOC runtime provides for several execution paradigms for object's and method invokation.

COOC is a superset of C, C++ and Objective-C. The underlying syntax of COOC is the same as that of Objective-C. The COOC compiler converts COOC into C

We present COOC's concurrency object model, runtime and support for various execution and service policies.

1 Introduction

Concurrent systems have been noted for their power [1]. For example, concurrent features are suitable for specifying and reasoning about systems in which several tasks or events may occur simultaneously: operating system; databases; and user interfaces (in particular multi-user systems, like groupware). Moreover the effective use of multiprocessor architectures requires algorithms and programs which are concurrent in nature, and thereby exploit concurrency as parallelism. However, while the benefits of concurrency are many, concurrency adds complexity to a system. Concurrent languages require additional syntax and semantics for specifying concurrent components; their interaction and control.

Object oriented programming consists of two inter-related themes; **encapsulation** and **inheritance** [7] [3].

Encapsulation data is not accessible directly, instead the supplier of data also has to provide the methods (interface) to manipulate it. These two components, the functions (procedures) and the data, make up an object. Basically, objects provide a method for abstracting functionality from implementation hence localising change and promoting development of systems via predefined units, or components [2]. The question of how an object is defined is addressed by the notion of a class. A class provides the means of defining those properties (services) that are generic to a set of similar objects, namely the functionality and data model. In this respect a class provides a template for defining individual objects.

Inheritance is a mechanism for sharing and version management of functionality and data. Instead of reimplementing/copying some system before extending or updating its functionality, inheritance provides for a dynamic sharing mechanism. Inheritance allows the new class, representing a variation of an existing abstract data type, to dynamically share common specifications from an existing class. Inheriting is preferred to static sharing, like copying, because if the superclass is subsequently extended or debugged, objects based on this superclass can automatically be updated.

Objects may provide a solution to both (complexity and concurrency) in that objects provide a logical unit which encapsulates data with functionality and communicates via message passing, while concurrent systems require a unit of computation which can take place largely independently of other computation and which have a mechanism for interaction. The introduction of concurrency to the object oriented model raises many questions [7]. It should be noted, however, that the encapsulation provided via object oriented systems affords an extension of the client server model; sending a message to another object can be seen as making a request for a service.

Concurrent Object Oriented C (COOC)[1] objects provide a generalisation of the client server paradigm. COOC employs a wait-when-necessary communication model and affords shared resource protection (via exclusive method invocation). The runtime for COOC provides a resource boundary, possessing a number of threads and a mail queue. Various scheduling and evaluation models can be supported by employing differing resource configurations. For example limiting the number of threads results in lazy evaluation, while limiting the mail queue size gives rise to a coarser grain.

Section 2 presents the COOC language: its unification of process and object; its client perspective, message sends and reply handling; and its server perspective, message receive and dispatching. Section 3 details the functionality of the COOC runtime and behaviour under various resources, threads and message queue, configurations. Finally in Section 5 we give our conclusions and areas of future research and development.

[1]The COOC language and execution model is realised on a C base: COOC is an extension to C. The underlying syntax of COOC is the same as that of Objective-C [2], moreover Objective-C objects are COOC objects which are to be evaluated with a sequential runtime and employ no concurrency control. Additional syntax provides for concurrency aspects. The current compiler converts COOC into C and supports C++ [4]. Hence COOC is a superset of C, C++ and Objective-C.

2 COOC Model

COOC is based on the view that notions of objects and aspects of concurrency could form a useful union. This union could result in a system in which desirable aspects of concurrency like modelling and speed-up can be enhanced while complex issues like control and synchronisation can be hidden or supported elegantly (within a message passing object paradigm).

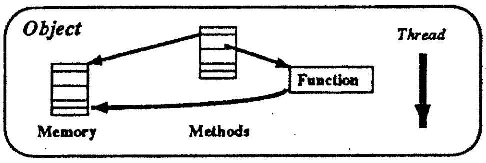

Figure 1: COOC Object

2.1 COOC Object Model

In a concurrent system a given object may receive a request from several clients at the same time. If several requests are processed concurrently they may interfere with each other and hence lead to buggy and unpredictable code. There are several approaches to resolving this problem: **atomic objects, exclusive methods, interface abstraction.**

Atomic objects are objects which can only be servicing one request at a time, subsequent requests are blocked until the current request has been completed. This approach, however, results in deadlock problems [8]; request sequences which contain the same object twice will block[2] and result in deadlock.

Exclusive methods extend the method interface to specify those methods that cannot be serviced concurrently. Methods that change the internal state of an object and so may interfere with each other's execution are specified as exclusive and sequentialised This model allows for an object to have internal concurrency, the servicing of several requests at the same time; hence resolving some of the deadlock problems associated with atomic objects. Deadlock still occurs if concurrent requests require resources to be changed.

Interface abstraction affords the programmer a meta-level, or control level, for specifying the concurrent interface to an object class; for example the exclusivity of methods. While this approach generalises both atomic objects and exclusive methods it results in another level of complexity with respect to requiring a meta-level language and inheritance of the control specification.

[2]In the case of direct request (requests to *self*) this problem can be avoided by not actually making another service request, but just evaluating the local function.

```
#import <cooc/Object.h>

@interface Account :  Object
int accountValue;

/*class methods*/
+new:  (int)initialValue;

/*instance methods*/
-(int)checkBalance;

/*instance exclusive methods*/
@exclusive
-(int)withdraw:  (int)value;
-(int)deposit:  (int)value;
@end
@end
```

Figure 2: account.h - bank account header file

COOC objects not only provide an encapsulation boundary but also serve to provide a process boundary. An object is made up of memory, hidden functions, an interface (methods/services) and a process. COOC employs exclusive methods for defining a control interface. Typically, methods that change the state of an object are exclusive, while methods that do not are non-exclusive. For example, figure 2 presents the header file for the bank account object; the methods `withdraw` and `deposit` both change the state of the internal variable `accountValue` and so have been declared exclusive. In terms of object specification COOC's syntax and inheritance model is an upwardly compatible extension of Objective-C, with one additional syntactic construct `@exclusive`.

2.2 COOC Client Model

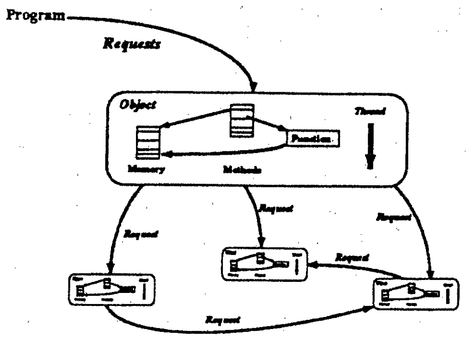

Figure 3: COOC language model

This section considers the client perspective of message requests. COOC objects communicate using message requests, *Figure 3*. Logically no control flow is transferred on a message send, however, as an implementation efficiency the client thread can be employed to support the service requested. Two communication models are provided to the client synchronous (*blocking*) or asynchronous (*non-blocking*).

Synchronous Synchronous messages require the sender to wait until the request has been fully serviced. While the basic synchronous communications model is uni-directional it can be extended to provide bi-directional information transfer, which is required for supporting a client-server relationship between two processes [1].

Asynchronous Asynchronous messages provide greatest flexibility, as messages can be sent and received in any order. This also increases concurrency in that message sends can be concurrent with other operations. However, using such features is difficult conceptually (as a message send may or may not have actually taken place) and practically (for example, debugging; as behaviour may be difficult to reproduce).

COOC employs a hybrid approach to synchronisation. On a message request the client is free to continue executing, hence the model is asynchronous, however, if and when the client requires the result the client process suspends and waits.

```
total = [bankaccount1 checkBalance]
  + [bankaccount2 checkBalance];
```

Figure 4: Synchronous messages in COOC

In the case where the result of a request is used just after the message send, *Figure 4*, the model is synchronous and bi-directional. The results of the two message sends are required immediately hence the client suspends awaiting both results.

```
(void) [bankaccount1 deposit:  10];
```

Figure 5: Asynchronous messages in COOC

In the case where the result of a request is not required, *Figure 5*, the model is asynchronous and uni-directional. This is true even if the result is not required under some condition.

```
balance1 = [bankaccount1 checkBalance];
...
balance2 = [bankaccount2 checkBalance];
...
total = balance1 + balance2;
```

Figure 6: Wait-when-necessary messages in COOC

In the case where the client requires the result some time after the message send, *Figure 6*, the model is asynchronous on send but synchronous on reply; known as **wait-when-necessary**. If the results from the message sends are available when required, the computation continues, if the results are not available then the computation suspends.

The base language for the COOC model is C, which allows for conditional branches and loops. This may result in several message sends being issued which have the same reply address. In these cases COOC assumes a **last sent** model: the reply should contain the result of the last message sent. Suspension and synchronisation is also based on the last message sent. Previous message sends are not terminated, however their reply is not returned by the runtime.

2.3 COOC Server Model

This section considers the server's, recipient, perspective of message sends in COOC. The execution of a process, its message sends and receives, its reads and writes and its use of various resources all require coordination and synchronisation. Several approaches could be employed for this synchronisation:

P/V semaphores The coordination of processes which employ shared resources, like memory or devices can be controlled through the use of semaphores. Before using a shared resource a process requests access to the resource by trying to gain access to a semaphore (this is known as a *P* operation). One the resource has been finished with it is released (by using a *V* operation).

Rendezvous To synchronise a client process with a server process a *rendezvous* mechanism can be employed. The client and server should meet at some point (period) in time, at which a request can be issued and, possibly, a reply obtained. This can be achieved by the client/server issuing an request/accept for a connection. If either process arrives at the rendezvous point before the other it blocks until the connection takes place. On rendezvous the client and server threads are synchronised and message passing and results can be transferred between processes.

Monitors A server process thread could be equipped with a mail queue which is *monitored* for certain type of requests. Monitors also define a set of interface procedures to which the monitor can respond, effectively providing an interface protocol for the monitor (those services the monitor can provide). For example if the server maintains a bounded buffer, when the buffer is full the server should field a **get** request next. Finally, a monitor is allowed to multiplex between several internal threads.

The concurrent access of encapsulated services forms part of the interface of an object, hence should also be encapsulated. COOC provides the exclusive method type to handle most cases in which resource synchronisation is required. If an exclusive method is currently being executed the subsequent exclusive requests will be queued and scheduled when the current method is finished. On certain occasions, however, an object may wish to effect the order of methods invocation, for example when a bounded buffer object becomes full. For this purpose COOC employs an abstract interface to the runtime; the runtime is encapsulated as an object.

```
- put:  (int) aValue
{
  buffer[next++] = aValue;
  if(next==bufffferMax)
    [[self runtime] scheduleNext:  get]
}
- get:  ...
```

Figure 7: Scheduling via COOC runtime

The local runtime for an object is obtained by a message `runtime` to `self`. The runtime method is defined in the class Object and so is inherited by all classes and objects. Figure 7 highlights how to request the runtime to schedule `get` once a bounded buffer is full.

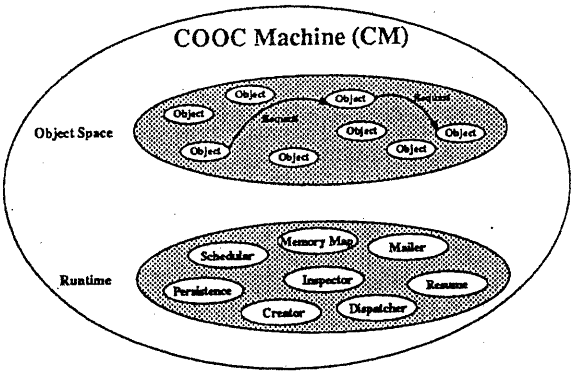

Figure 8: COOC Machine (CM)

The combination of an object space, all objects with the same runtime, and the runtime is known as a COOC machine *Figure 8*. A COOC machine provides a resource allocation boundary, a logical machine boundary and a policy boundary. In the network implementation the COOC machine provides the means by which objects are grouped and located.

3 COOC Runtime

The runtime has two resource parameters, {**number of threads** and {**mail queue size**. Varying these parameters allows for differing execution models and policies (see Table 1): the mapping and scheduling of requests depends on the available resource.

Mail Queue	Thread	Behaviour
0	1	Sequential (Obj-C)
small	+1	Course & Eager
large	+1	Course & Lazy
infinite	+1	Pure Lazy
large	large	Fine Grain

Table 1: COOC runtime execution models

3.1 Message Send

A message send is compiled into a function call:

```
COOC_Send(id obj, char *selector,
void *reply_addr, int reply_size, args);
```
[3]

This function serves two purposes: firstly, house keeping related to message loops, conditional and reply handling; and secondly, to either place the message in the runtime mail queue or dispatch the message.

3.1.1 House Keeping

Firstly, as noted in section 2.2, it is possible for several message sends to have the same reply address, in this case the runtime should return the result of the last message send. To achieve this a record of each message send and whether a reply is required is noted. If another message with the same return address is requested, the runtime invalidates the required flag of the previous message send and then creates a new send record. On completion of a request the runtime checks if the result is still required before returning its value to the client.

Secondly, reply addresses that become invalid, for example at the end of a program block, are invalidated in the runtime by a compiler inserted function:

```
COOC_Clear(void *reply_addr);
```

3.1.2 Queue or Dispatch

A message send corresponds to placing the message in a runtime mail queue, the client object is then free to continue execution. If the mail queue is full the runtime does not make a record of the request as indicated; instead the request is dispatched for immediate evaluation and employs the client's thread (as in a function call). Hence, when the message queue becomes full the current implementation defaults to a sequential evaluation, in which a message send corresponds to an indirect function call.

Note that in the limit, if the mail queue size is specified as zero, the execution defaults to a sequential model (Objective-C).

3.2 Message Reply

COOC employ's a wait-when-necessary model (*see Section 3*) hence before the result of a message send is used the system, compiler and runtime check if the result is available. To this end before the result is used the compiler inserts a call to the runtime:

```
COOC_ReplyWait(void *reply_addr);
```

3.2.1 Suspend or Continue

If the request has been completed and the result obtained the client's execution continues. However, if the result is pending the scheduler is notified to prevent further execution of the client until the result is available.

3.2.2 Eager or Lazy

In the case where the request has not been dispatched, the scheduler immediately dispatches the request and employs the client's thread to perform the evaluation. This gives rise to a lazy evaluation execution model. In the limit where the mail queue size is infinite and only one thread is employed the model is purely lazy; only required results are evaluated.

[3]obj is the recipient object's id; selector is the method selector; reply_addr is the reply address; reply_size is the size of the reply type; and args are the method arguments.

4 Example-Shared Draw

Shared Draw is a groupware application implemented in COOC [6]. Each member of the group manipulate a drawing via a local graphical interface, as is found in a conventional drawing program. However, the elements that make up the drawing are not stored locally, but are accessed via shared memory (in the C implementation) and via a database object (in COOC implementation). In this section we consider the concurrency control aspect of this application by considering pseudo which employs C with lightweight processes and COOC.

```
main(){
  :
  lwp_setstkcache(STACK_SIZE, THREAD_N);
  lwp_create(0, scheduler, SCHPRIO,
      0, lwp_newstk(),0);
      :
  while(i < THREAD_N) {
    if (new_user(name, &dpy)) {
      lwp_create(&newthreadId,
        UI, MINPRIO, 0, lwp_newstk(),
        1, dpy);
      i++;
    }
    else
      lwp_yield(THREADNULL);
  }
}

scheduler(){
  struct timeval interval;

  interval.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&interval);
    lwp_resched(MINPRIO);
  }
}

aUItask(dpy){
  :
  lwp_setpri(SELF, MAXPRIO);
    /* manipulate global data */
  lwp_setpri(SELF, MINPRIO);
    :
}
```

Figure 9: Shared Draw - toplevel in C

Figure 9 presents pseudo code for Shared Draw implemented in C. The system first creates a set of stacks to be used by each lightweight thread. It then creates a scheduler to provide for the time slicing of threads. Finally, while threads are still available a new interface is created for each new user. The interface threads invoke an X-based drawing interface and behaves like a any other window application. Using this model shared information is maintained as global data, functions that manipulate this data, **aUItask**, such as callbacks from an individuals interface are required to provide for critical regions: in this case by directly employing process priorities (other models - semaphores, monitors, etc - could also be used).

```
main()
{
  db = [DBClass new];
  um = [UMClass new];

  while(1) {
    if (new_user(name, &dpy))
      [um add:name display:dpy db:db];
  }
}
```

Figure 10: Shared Draw - toplevel in COOC

Figure 10 presents pseudo code for Shared Draw implemented in COOC. Concurrency is maintained by the underlying runtime system, so the programmer just creates objects which can act concurrently. The system first creates a databased object for maintaining the shared drawing information[4] (objects). Then a user manager object is created which provides for subsequent creation of user interface objects. Each user interface object is informed of the id of the database object at create time, this id is then employed by methods which required access to the shared information. Protection of the shared information is maintained via exclusive methods being employed in by the databased objects interface. Process and memory allocation for each concurrent activity are supported by the runtime system.

[4] Currently COOC does not directly support databased functionality, as in an ODBMS. Hence a database object is employed to control access and updates of the shared objects.

5 Conclusions

COOC is based on the premise that notions of objects and the aspects of concurrency form a useful union. This union results in a system in which desirable aspects of concurrency can be enhanced while complex communications issues can be supported elegantly.

The basic specification of COOC objects, classes and inheritance, is the same as for Objective-C. However, COOC introduces the notion of exclusive and non-exclusive methods. This provides for cases in which methods change state and hence may effect the execution of other method invocations. In certain applications, such as a bounded buffer, finer level control of the scheduling of requests may be required, to this end COOC affords an encapsulated runtime interface. The client of a message send employs a wait-when-necessary model; this allows both asynchronous and synchronous communication to be easily realised. Furthermore, communications can be uni- or bi-directional.

The COOC runtime currently has two resource parameters; mail queue size and number of threads. If the mail queue becomes full a message send defaults to an indirect function call, as in current sequential implementation. In the case where the mail queue size is specified as zero COOC behaves like Objective-C and can be used to execute Objective-C code; in this case all the methods are non-exclusive. When the result of a message send is required a call is made to the runtime. If the result is available the execution continues, if the result is pending the runtime suspends the further evaluation of the client. In the case where the request has not as yet been allocated a thread the client's thread is used. In the limit, when only one thread is available, the evaluation become lazy; only when a result is required is the message request evaluated.

We have realised the COOC runtime using both lightweight processes and sockets available on SUN's SPARC based workstations. A prototype compiler has been realised, which also supports cross compilation of C, C++, Objective-C and COOC; we hope to make a beta release available in the near future. The compiler converts COOC code into C which is then compiled using a standard C compiler.

The runtime also supports a multi-layered debugging environment [5], which consists of four components: a high level object and message oriented tracer; an object-thread level (internal concurrency) tracer; a thread-process level (dbx) tracer; and a class browser. These different tracers have been integrated into one uniform environment which allows the programmer to control and view concurrent execution at different levels.

References

[1] G. R. Andrews and F. B Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surverys*, 1983.

[2] B. J. Cox. *Object Oriented Programming - An Evolutionary Approach.* Addison-Wesley, 1987.

[3] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988. Series in Computer Science.

[4] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[5] R. Trehan, N Sawashima, and K Maeda. Concurrent Object Oriented 'C' (COOC) Debugging Enviroment. In *Workshop on Future Directions of Parallel Programming and Architecture, International Conference on Fifth Generation Computer System*, 1992.

[6] R. Trehan, N. Sawashima, and A. Morishita. CSCW Architecture and Shared Drawing Tool Example. In *The Eighth Symposium on Human Interface*, 1992. Submitted.

[7] P. Wegner. Concepts and Paradigms of Object Oriented Programming. In *Proceedings of OOPSLA*, 1989. Keynote Talk.

[8] Y. Yasuhiko and M. Tokoro. Concurrent Programming in Concurrent Smalltalk. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming.* MIT Press, 1987.