Technical Report

UCAM-CL-TR-76
ISSN 1476-2986

Number 76

Preserving abstraction in concurrent programming

R.C.B. Cooper, K.G. Hamilton

August 1985

ISSN 1476-2986

Preserving Abstraction in Concurrent Programming

R C B Cooper and K G Hamilton

University of Cambridge Computer Laboratory

Index Terms: abstraction, concurrency, deadlock, interface, module, monitor, operating system, process, synchronization

Abstract.

Recent programming languages have attempted to provide support for concurrency and for modular programming based on abstract interfaces. Building on our experience of adding monitors to CLU, a language orientated towards data abstraction, we explain how these two goals conflict. In particular we discuss the clash between conventional views on interface abstraction and the programming style required for avoiding monitor deadlock. We argue that the best compromise between these goals is a combination of a fine grain locking mechanism together with a method for explicitly defining concurrency properties for selected interfaces.

1. Introduction.

Modular programming has received much attention as a way to control the complexities of large programs. A number of recent programming languages have provided direct support for defining modules with type-checked external interfaces which hide the underlying code and data implementions. Similarly the complexities of concurrent programming have been addressed by a number of different programming techniques and language features such as monitors, message passing and Ada rendezvous.

The problem examined in this paper arises when these two themes are combined when writing large concurrent systems. Simply stated, the problem is a serious conflict between interface abstraction and certain requirements for concurrent programming – particularly that of avoiding deadlock.

In a concurrent program, aspects of a module's implementation may compromise the correct functioning of its clients. Traditionally these aspects are not expressed in the interface which defines the external view of that module as used by its clients. They include the use, or lack, of concurrency in the implementation, as well as the inter-relationship of the implementation with other modules.

We encountered this problem in our work in the Mayflower group at Cambridge. The group was founded in 1982 to build an environment for distributed applications. One of our first needs was a suitable programming language to be used both for implementating our system and as the language in which applications would be programmed. Concurrency is arguably the major cause of difficulty in distributed and systems programming, and it ought to benefit from direct support in a high-level language.

After considering a number of languages, we eventually selected CLU [6] as our base language, and undertook to add concurrency to it ourselves, in the form of monitors. CLU combines strict data typing with an extremely consistent view of user-defined abstract data types, providing good support for a modular programming style. It is CLU's emphasis on data abstraction which has highlighted the problem for us. But this problem is not unique to CLU – it will occur in any language which tries to hide implementations behind abstract interfaces. The methodology we present for handling this problem is applicable to other languages.

We chose monitors in preference to a message passing system because they fitted in well with CLU's procedural orientation and could be implemented efficiently.

As a result of this decision, our discussion is orientated towards monitors, although the problem we discuss also occurs in message passing systems.

2. The Problem in Detail.

In this section we review the requirements of concurrent programming and how monitors are intended to support these. We show how effective use of monitors requires knowledge of certain aspects of the implementations of the modules they use, and how this conflicts with abstraction.

2.1 Requirements for Concurrent Programs.

There are at least three requirements for any concurrent program on top of those of purely sequential programs:

1. There should be no interference between processes. That is, there should be no concurrent use of data structures which do not have well-defined behaviour under sharing. There should be no violation of data structure invariants and no viewing of data in an inconsistent state.

2. There should be no deadlock. The definition of what constitutes deadlock is problematic. It does not only involve processes waiting indefinitely on semaphores or monitor locks for which there are no potential notifiers. Lack of progress or livelock, in which processes cycle indefinitely never making progress on their tasks, is just as serious.

3. There are efficiency and effectiveness criteria which, although less well defined than the previous two requirements, are often crucial to a program's success. It is particularly important in some applications to avoid unnecessary serialization or exclusion. In real-time applications, such as the implementation of network protocols, effective performance constitutes a correctness criterion.

In the monitor of Brinch Hansen [1] and Hoare [4] the solution to requirement 1, above, is to restrict process interactions to within monitors and to enforce mutual exclusion on all the procedures of a monitor. This *classical* approach, as we term it, is impractical in large systems since it tends to increase possibilities of deadlock and reduce performance (violations of requirements 2 and 3). The likelihood of deadlock is the more serious problem and warrants examination.

2.2 Monitor-Induced Deadlock.

Undoubtedly a significant number of deadlocks in concurrent programs are the result of inadequate algorithm design, or of applications in which unpredictable contention for resources occurs. We make a terminological distinction here between resources, which are shared objects of significance in the application (often representing real-world objects), and the mechanisms used for low-level synchronization. It is the possibility of deadlock arising from low-level synchronization with monitor locks, that concerns us, since it would be unfortunate if the language features intended to ease the programming of concurrent systems were themselves the cause of program faults.

2.2.1 Module Recursion.

The simplest form of monitor-induced deadlock involves only a single process which attempts to claim for a second time a monitor lock which it already holds. This can occur when a procedure of a monitor calls some other procedure which eventually calls back into the original monitor.

A deadlock involving more than one process can occur if two monitors, M and N, are structured so that a procedure of M, holding its monitor lock, calls a procedure of N and tries to claim N's lock, while a procedure of N calls a procedure of M in the same way. It is possible for two processes, by calling these procedures, to each hold one of the monitor locks while trying to claim the other. Note that the processes are not deadlocking over *resources* but merely over *synchronization*. This case may arise even when there is no fundamental resource allocation deadlock in the application.

Both these forms of deadlock could be avoided by forbidding *mutually recursive modules*. Module recursion occurs when modules use each other in a cyclic way.

Module recursion can be avoided by organizing modules into a hierarchy, so that procedure calls only flow down from higher level abstractions to lower levels. In many situations this is not the natural structure. Small groups of modules may be inter-dependent and would have to be combined into a single monolithic module to preserve the hierarchy. Also procedure variables provide a useful interaction mechanism between different levels of an application and would probably have to be forbidden if a hierachical structure was to be enforced.

2.2.2 Nested Monitor Calls.

Another case of deadlock occurs even if module recursion is not present. This arises when a process P calls into a monitor M, which then calls into a monitor N, claiming both monitor locks in doing so. P now suspends inside the nested monitor N waiting for some condition to become true. While it is waiting it releases N's lock but not M's. If some other process Q, which will establish the condition, has to execute a similar sequence of monitor calls, deadlock will result. This is so because the waiting process P still holds the lock for M, which Q must claim.

Even if the conditions for deadlock do not occur, a suspended process holding a monitor lock will prevent other processes from executing in that monitor until the suspension has finished. In some cases, such as suspension awaiting input on a network port, the delay may be arbitrarily long.

This problem has been discussed in the literature as the nested monitor call problem [7], and some writers have proposed that all monitor locks held by a process should be released when it suspends inside a monitor. This is not a satisfactory solution as it would require monitor data to be maintained in a consistent state prior to monitor calls, and indeed prior to every procedure call (since any procedure might call into a monitor).

Equally, banning calls into nested monitors is impractical, and would prevent hierachical design.

2.3 Message Passing.

It is interesting to consider if these problems with monitors could be avoided by using the message passing model when structuring concurrent programs. In particular modules could use a *send no-wait* message to request actions of other modules, and then wait for any messages – both replies to outstanding requests and further incoming request messages. This would avoid a module being unable to service further clients while waiting for results from other modules.

Unfortunately this suffers from a problem similar to that encountered if all monitor locks are released upon process suspension, namely that the module data invariants must be restored prior to any potential message exchange so that other client requests will not see the data in an inconsistent state.

2.4 Relaxing Mutual Exclusion in Monitors.

We have discussed problems with the classical monitor model. A more flexible approach is taken with monitors in the Mesa language [5]. Here procedures in a monitor module not requiring mutual exclusion throughout need not claim the monitor lock when they are called. These procedures may call other, mutually excluded, procedures in the monitor to perform activities that must be excluded in time.

This kind of monitor was the model for our initial work on adding concurrency to CLU. Relaxing the requirement of mutual exclusion on all monitor procedures should reduce the possibilities of deadlock.

Based on the analysis of deadlocks above, there are two classes of procedures whose use is to be avoided while holding a monitor lock. Firstly, there are procedures which result in module recursion; and secondly procedures which suspend themselves on semaphores, or otherwise delay for significant periods.

However identifying those procedures requires information about other modules which traditionally appears nowhere in their interface specifications. To see why this is important we will look at what modular programming entails.

2.5 Modular Programming.

Modular programming is a style in which sets of procedures along with the data they operate on are encapsulated in modules. Viewed from the outside each module is defined in terms of its external interface – the procedures (and in some models the data) which other modules may use. How each of the procedures is implemented and what private data structures exist should be of no concern to clients of the module. Modular programming is given varying levels of support in such languages as Mesa [8] and Ada [9].

Modular programming practices allow implementors the freedom to consider different techniques in the implementation of their modules – and to perhaps change these decisions later. Clients of these modules are made immune from these implementation considerations and spared unnecessary detail about the internals of a module. It should be possible to read and understand modules without needing information about other modules apart from that which appears in their abstract interfaces.

Using a modular programming style requires much discipline on the part of the programmer, even in a language like CLU which directly supports modularity. While clients of an abstraction are prevented from actually accessing the representation of an abstract type, there is nothing preventing the use of *knowledge* about the implementation and how it is coded. This is a particular problem when one person is both implementor and client of the same abstraction. Personal experience has demonstrated that "bugs" in the specification of interfaces often do not manifest themselves until abstractions are modified or re-used by different programmers.

2.6 Discussion.

The concept of modular programming is too important to lose. It is about the only way we know to build and maintain large software systems, especially those programmed by more than one person. Determining whether the procedures of a module perform semaphore waits, checking what other modules they use, etc., violates modular programming. Is there a viable compromise between the two?

3. Towards a Modular and Concurrent Programming Style.

We have described a fundamental contradiction between two desirable programming goals: avoiding deadlocks and hiding details of modules' implementations. To cope with this conflict we present a methodology in which some information, chiefly about concurrency aspects of module implementations, is included in interfaces, but in a way which preserves a large measure of abstraction. This is combined with guidelines for coding client modules, intended to limit the number of modules whose concurrency properties must be known.

3.1 Fine Grain Mutual Exclusion.

The first aim of this style is to minimize the amount of processing that is performed while holding the monitor lock, so as to reduce the scope for deadlock. Code inside a monitor is structured so that any lengthy processing which does not require mutual exclusion throughout is performed in unlocked procedures, which call short internal procedures to perform those actions that must be excluded.

This approach is useful because much of the code in a monitor may only require occasional access to shared data structures and may principally deal with objects that are owned by a single process. For example a procedure may not require mutual exclusion while processing its argument objects or while processing objects which have been removed from shared data structures. Additionally, some operations on the monitor's data, particularly simple read operations, may not require mutual exclusion. Minimizing excluded code in this way can greatly reduce the number of procedures whose concurrency properties affect a monitor's behaviour, because most procedures are called while no monitor lock is held.

3.2 Assigning Concurrency Properties to Interfaces.

All operations in CLU have the semantics of procedure calls, including those of built-in types such as integer addition and array element access. To perform any action, mutually excluded code must therefore include procedure calls on abstract interfaces. To avoid deadlock the monitor's implementor must ensure that these interfaces have reliable concurrency behaviour. Equally, the implementor must ensure the safety of any operations performed on shared data without mutual excusion.

In our experience, the information required to ensure this can be expressed by two concurrency related properties associated with interfaces. One or both of the following properties must be attached to a subset of the operations of CLU's built-in and user defined abstract types.

1. *Internally synchronous.* This property applies to procedures which can be called safely by concurrent processes without any external synchronization, e.g. because they use mutual exclusion internally.

   CLU defines a set of *immutable* types whose values, once created, cannot be changed. Thus even though their operations are not mutually excluded, all the operations on these types are internally synchronous. More subtly, among mutable types some operations are known to be internally synchronous while others are not.

2. *Restricted.* A procedure is deemed restricted if its implementation refrains from suspending itself, from calling a procedure with the potential to cause module recursion, or from calling any non-restricted procedure.

   It is useful to relax this definition slightly. For instance, some procedures may wish to call a debugging procedure if an internal consistency check fails. If the debugging procedure suspends itself (e.g. to perform I/O) then its client ought to be non-restricted, but it may be more sensible to categorize it as restricted.

Unfortunately, determining the class of a module requires determining the classes of the transitive closure of all the modules used in its implementation. Even worse, changes in the interface of a low-level abstraction logically require re-examination of every module which uses it, including indirect use through several other abstractions. These concurrency facets, unlike any other facets of interfaces, can have effects across multiple abstraction boundaries.

It has only been necessary to define concurrency properties for a subset of CLU's built-in types and for a small number of user defined abstract types. In many of these cases the concurrency properties were already quite obvious. For example, the integer addition operation could reasonably be expected not to have undesirable concurrency properties. Contrariwise, it is unsurprising that the stream clusters's input and output operations are non-restricted, since they may have to wait for I/O completion. However, due to their implementations, some interfaces did have undesirable concurrency properties which were not self-evident.

Note that the classification is strictly on the *operations* of a module, not on the whole module. So some modules may have operations which fall into different classes, although this is not common.

3.4 Discussion.

To some degree, this style represents a defeat for modularity. But at least those aspects of the implementation on which clients depend have become explicit in the contract which the abstract interface represents between client and implementor. In our programming in the past, assumptions about the concurrency aspects of modules have always been made. Occasionally these assumptions were wrong. In other cases we simply read the code of the appropriate implementations to find information not present in the interface. We are sure most programmers have done this. The main contribution of the style we are proposing is to recognize this kind of implementation dependency and control it.

4. Support from the Language.

Earlier we expressed our belief that direct support for concurrency should be provided in the programming language. We do not believe a language can, nor should enforce a fixed style of concurrent programming. Rather it should provide *tools* which encourage the construction of safe and effective programs. This section describes how we have tried to achieve this in our work with Concurrent CLU.

4.1 Concurrent CLU with monitors.

CLU's principle data abstraction facility is the *cluster*, which provides a structuring facility somewhat like the Simula *class*. Each cluster defines an abstract type consisting of a set of operations (essentially procedures) which may be performed on a concrete representation type. The representation is never visible outside its defining cluster.

Our initial version of Concurrent CLU incorporated monitors as a special kind of cluster. Procedures of these monitor clusters which wanted to execute under mutual exclusion, could identify in their procedure headings which monitor lock they wished to claim. This explicit monitor lock naming was most useful for per-object locking, in which one lock is associated with each object of the monitor's type. This permitted procedures of the monitor to execute concurrently, where appropriate, when they were operating on different objects.

We also included semaphores for synchronization between different processes in a monitor, and provided processes which could be created dynamically.

We used Concurrent CLU, in this original form, to implement an operating system supervisor, including a number of communication protocol drivers and an interface to a remote file system; a remote procedure call system; and a source level remote debugger. Concurrent CLU has been used by others to develop a network resource management service [2]; a controller for an integrated voice/data project; and a parallel processing worm system. It is also being used in the development of a distributed compilation system.

As we used Concurrent CLU our ideas on concurrent programming changed, culminating in the programming style presented in this paper. Our programming practice had outgrown the initial set of concurrency features we had added to CLU. We considered how best to change Concurrent CLU to support our needs, and decided to replace monitors by a finer-grained mutual exclusion mechanism, *gates*. We have considered, but not implemented, language features for specifying concurrency properties of procedure interfaces.

4.2 Revising Concurrent CLU.

Monitors do not encourage the programmer to minimize the use of mutual exclusion. Frequently sections of code requiring exclusion consist of only a few lines, performing some simple update on the monitor's state. Unfortunately these few lines of code often do not constitute any obvious logical operation. This leads either to monitors containing a series of obscurely named, fragmentary procedures, or to the use of monitored procedures which hold the lock over more operations than is strictly necessary, so that they may constitute a clear logical sub-unit of the module. In order to improve program clarity we tended to do the latter, even though it increased the scope for deadlock.

On reflection the classical monitor, and to a lesser degree the Mesa monitor, were wrong in the way they combined procedural abstraction and mutual exclusion in the one feature: monitor procedures. For our programming style, a return to a syntax more akin to Hoare's critical regions [3] was desirable.

What was required was a locking mechanism for blocks of code smaller than procedures. However, to generalize the locking mechanism, we also wished to allow implementors to define new locking disciplines for their own abstract types. This would permit the definition both of new locking abstractions for general use, such as a distributed lock manager for use in multi-machine applications, and also application specific locking disciplines for individual abstract types. For example, some abstract types might choose to implement a form of read locking, so that several processes can perform read operations simultaneously, while excluding any update operations. If application specific locking operations are permitted locking can cease to be just a low-level synchronization operation and can involve long term scheduling decisions. Such a generalization of the locking mechanism appears in keeping with the spirit of abstraction languages such as CLU.

We therefore developed a new form of control abstraction for Concurrent CLU, the *gate*. This executes in two halves, one to establish the lock condition and one to clear it. A gate definition is very similar to a CLU procedure definition with the keywords **key** and **keys** used in place of **return** and **returns**. (Gates are in fact more closely related to another of CLU's control abstractions, *iterators,* than to conventional procedures.) A much simplified example of a gate definition appears in figure 1. A gate definition would typically be part of a cluster implementing an abstract type. To present a more representative example, however, would require explanation of CLU's syntax for abstract type definition.

```
claim = gate (sem: semaphore, timeout: int) signals (timed_out)
    % Use a semaphore to establish mutual exclusion. If the semaphore
    % cannot be claimed within the specified time raise an exception.
    semaphore$wait (sem, timeout) resignal timed_out
    % Allow our client's using block to execute.
    key
    % Now our client is finished, release mutual exclusion.
    semaphore$notify (sem)
end claim
```

Figure 1. A gate definition.

```
mutex: semaphore: = semaphore$create(1)
    % Create and initialize a semaphore to use as a mutual exclusion lock.
....
% Claim the lock, timing out after 5 seconds if it is in use.
using claim (mutex, 5) do
    % Statements here execute under mutual exclusion from other uses of
    % this lock.
    ....
end
....
```

Figure 2. Use of a gate.

A gate is invoked by a **using** statement which contains a block of code (see figure 2.). When the **using** statement is executed the gate is entered. Execution continues until a **key** statement is encountered, at which point control temporarily returns to the calling procedure and the code block of the **using** statement is executed. When this code block is terminated for any reason (e.g. by raising an exception, executing a procedure **return** statement, or simply reaching the end of the **using** block) the gate is resumed from just after the **key** statement. When the gate terminates, execution continues after the **using** statement. Thus a gate is executed in two parts: a prelude which may claim a lock and a sequel which may free it. A **key** statement may return a set of results to be used by the gated block.

The main point of gates is that they provide synchronization on small units of code. We have chosen to make them more general than simple critical regions in keeping with CLU's existing facilities for user-defined types. However, it is the overall aim of gates that is important, rather than the details of their integration into a particular language. The provision of a synchronization mechanism which allows fine-grain locking encourages programming in a manner which reduces unnecessary mutual exclusion, reducing the opportunity for creating deadlocks. In some cases it may also improve performance by permitting extra concurrency.

4.3 Putting Concurrency Properties into Interface Types.

While we have defined concurrency properties for a number of CLU interfaces, this has been an informal process which has received no language support. The concurrency attributes of an interface only have the status of comments. Concurrency properties are sufficiently important to merit formal inclusion in interface definitions. We have not made these extensions to CLU, partly because we did not wish to change CLU's type system and partly because they did not fit cleanly into CLU's existing module structure.

The first step is clear. Procedure interface definitions should have the option of specifying whether they are synchronous or restricted as part of their types. Interfaces which fail to specify either must be assumed to be both non-synchronous and non-restricted. Procedure variables must also be typed with these concurrency properties, and non-synchronous or non-restricted procedures must not be assigned to synchronous and restricted variables, though the reverse would be permitted.

This solves one half of the problem, specifying the concurrency contract in the definition of the interface. Specifying the contract at uses of the abstraction is harder. Having to specify concurrency properties on individual procedure calls would be irksome, particularly as this would interfere with several pieces of syntactic sugar that CLU provides for accessing record and array structured objects. Instead, it seems desirable to centralize this interface information as part of an explicit contract between a module and the outside world. CLU lacks the explicit *import lists* of languages such as Mesa [8] and Modula-2 [10]. Such import lists are an appropriate way of defining a module's external dependencies, including such things as concurrency properties. Thus an interface might be imported with the attributes synchronous or restricted.

During compilation the client's desired concurrency properties can be checked against those of the implementing module and any discrepancy reported. In particular, if a procedure's interface properties change, this will be detectable at compile or link time via the normal interface consistency checking.

Thus programmers may be confident that interfaces they are using have declared the concurrency properties they desire. However there is no guarantee that the body of a procedure actually conforms to its defined concurrency properties. This is considerably more difficult to establish than, say, checking the type of a result object. For example, in the case of the synchronous property, the correctness of this property for one operation on an abstract object may depend upon the behaviour of other operations that may be called concurrently. Also, the restricted property may be relaxed occasionally, as in the example in section 3.2 of a non-restricted debugging routine which is almost never called. (It might, however, be useful for the compiler to issue a warning message in cases where a restricted procedure calls a procedure which has not been imported as restricted.) It appears that the correctness of concurrency properties must be left to implementors. While far from ideal, this is not unreasonable, given that many existing interfaces have well known semantic properties that cannot be checked by the type system. For instance if an abstract type supports *add* and *equal* operations, practical compilers are unable to ensure these operations exhibit the expected commutative and side-effect free semantics.

On the client side it might appear desirable to enforce some restriction such as forbidding non-restricted calls within gated regions. However, there may be legitimate reasons for such calls, particularly as user defined gates may define more complex locking properties than simple mutual exclusion. Once again, a compiler warning message seems appropriate.

5. Conclusions.

We have described how the goal of interface abstraction clashes with the requirements of concurrent programming. We have taken two measures in dealing with this conflict. Firstly fine grain locking minimizes the amount of code affected by the conflict; and secondly concurrency information is specified as part of the interface of certain modules. Large monitor based systems can be developed without these techniques, but their use makes implementation easier and leads to a sytem which can be more easily modified.

Making concurrency properties part of interfaces' types can be regarded as a logical extension of normal type-checking. Ideally we would like to describe all of a module's externally visible behaviour in its interface types, but much of this behaviour may be very hard to define. The concurrency properties we have defined are particularly good candidates for inclusion in interfaces because of their relative simplicity, combined with their importance for program correctness.

Acknowledgements.

We thank the other members of the Mayflower Group who have contributed to the experience on which this work is based. Particular thanks are due to Andrew Seaborne for implementing the gate concept. We are indebted to Dan Craft, Roger Needham, and Julian Pardoe who carefully read and commented upon earlier drafts of this paper.

References.

[1] P. Brinch Hansen, *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] D. H. Craft, "Resource Management in a Decentralized System", *Proc. 9th ACM Symposium on Operating Systems Principles* (*Operating Systems Review*, **17**, 5), pp 11-19, Oct 1983.

[3] C. A. R. Hoare, "Towards a theory of parallel programming" In C. A. R. Hoare and R. H. Perrott (Eds.), *Operating Systems Techniques*, pp 61-71. Academic Press, New York, 1972.

[4] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, **17**, 10, pp 549-557, Oct 1974.

[5] B. W. Lampson, D. D. Redell, "Experience with Processes and Monitors in Mesa", *Communications of the ACM*, **23**, 2, pp 105-117, Feb 1980.

[6] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *CLU Reference Manual*, Vol. 114, *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[7] A. M. Lister, "The Problem of Nested Monitor Calls", *Operating Systems Review*, **11**, 3, pp 5-7, July 1977.

[8] J. G. Mitchell, W. Maybury, R. Sweet, *Mesa Language Manual*. Xerox PARC, Palo Alto, Calif., April 1979.

[9] U.S. Department of Defence, *ADA Programming Language*, ANSI / MIL-STD-1815A. Washington D.C., 1983.

[10] N. Wirth, "Modula-2", Technical Report 36. ETH Zurich, March 1980.