μC++: Concurrency in the Object-Oriented Language C++

P. A. Buhr*, Glen Ditchfield*, R. A. Stroobosscher*, B. M. Younger*, C. R. Zarnke**

* Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1
(519) 888-4453, Fax: (519) 885-1208, E-mail: pabuhr@watmsg.uwaterloo.ca

** Hayes Canada Inc., 175 Columbia St. W., Waterloo, Ontario, Canada, N2L 5Z5

SUMMARY

We present a design, including its motivation, for introducing concurrency into C++. The design work is based on a set of requirements and elementary execution properties that generate a corresponding set of programming language constructs needed to express concurrency. The new constructs continue to support object-oriented facilities like inheritance and code reuse. Features that allow flexibility in accepting and subsequently postponing servicing of requests are provided. Currently, a major portion of the design is implemented, supporting concurrent programs on shared-memory uniprocessor and multiprocessor computers.

KEY WORDS: Concurrency, C++, Class-based Objects, Postponing Requests

INTRODUCTION

The goal of this work is to introduce concurrency into the object-oriented language C++ [1]. To achieve this goal a set of important programming language abstractions were adapted to C++, producing a new dialect, which we have called μC++. These abstractions were derived from a set of design requirements and combinations of elementary execution properties, different combinations of which categorized existing programming language abstractions and suggested new ones. The set of important abstractions contains those needed to express concurrency, as well as some that are not directly related to concurrency. Therefore, while the focus of the paper is on concurrency, all the abstractions produced from the elementary properties are discussed. While we are implementing our ideas as an extension to C++, the requirements and elementary properties are generally applicable to other object-oriented languages such as Eiffel [2], Simula [3] and Trellis/Owl [4].

Combinations of the elementary execution properties produced four abstractions: coroutine, monitor, coroutine-monitor, and task. Each of these abstractions was introduced into C++ through new type specifier that produced new kinds of objects. As well, other new constructs were introduced to control interactions among the new objects. A detailed examination is made of how inheritance affects these constructs. Several comparisons are made with concurrency-library schemes for C++ and other concurrent programming languages.

In general, casting the new constructs into a concrete syntax for a new programming language should have been straightforward. However, because we added the constructs to an existing language, C++, we did not have complete freedom of design, but had to conform to C++'s existing design. Blending each construct with the syntax, the semantics, and the philosophy of C++was an interesting engineering exercise. The selection of C++ was driven by pragmatic reasons—C++ already supports a general object creation facility, the class, on which we could base other kinds of object creation facilities, and C++ is becoming popular and readily available.

μC++ executes on uniprocessor and multiprocessor shared-memory computers. On a uniprocessor, concurrency is achieved by interleaving execution to give the appearance of parallel execution. On a multiprocessor computer, concurrency is accomplished by a combination of interleaved execution and true parallel execution. Further, μC++ uses a **single-memory model**. This single memory may be the address space of a single UNIX process or a memory shared among a set of UNIX processes. A memory is populated by routine activations, class-objects, coroutines, monitors, coroutine-monitors and concurrently executing tasks, all of which have the same addressing scheme for accessing the memory. Because these entities use the same memory they can be **light-weight**, so there is a low execution cost for creating, maintaining and communicating among them. This has its advantages as well as its disadvantages.

Communicating objects do not have to send large data structures back and forth, but can simply pass pointers to data structures. However, this technique does not lend itself to a distributed environment with separate address-spaces. Currently, we are looking at the approaches taken by distributed shared-memory systems to see if they provide the necessary implementation mechanisms to make the non-shared memory case similar to the shared-memory case.

DESIGN REQUIREMENTS

The following requirements directed this work:

- All communication among the new kinds of objects must be statically type checkable. We believe that static type checking is essential for early detection of errors and efficient code generation. (As well, this requirement is consistent with the fact that C++ is a statically typed programming language.)
- Interaction between the different kinds of objects should be possible, and in particular, interaction between concurrent objects, called tasks, should be possible. This allows a programmer to choose the kind of object best suited to the particular problem without having to cope with communication restrictions.

  This is in contrast to schemes where some objects, such as tasks, can only interact indirectly through another non-task object. For example, many programming languages that support monitors [5, 6, 7] require that all communication among tasks be done indirectly through a monitor; similarly, the Linda system [8] requires that all communication take place through one or possibly a small number of tuple spaces. This increases the number of objects in the system; more objects consume more system resources, which slows the system. As well, communication among tasks is slowed because of additional synchronization and data transfers with the intermediate object.
- All communication between objects is performed using routine calls; data is transmitted by passing arguments to parameters and results are returned as the value of the routine call. We believe it is confusing to have additional forms of communication in a language, such as message passing, message queues, or communication ports.
- Any of the new kinds of objects should have the same declaration scopes and lifetimes as existing objects. That is, any object can be declared at program startup, during routine and block activation, and on demand during execution, using a new operator.
- All mutual exclusion must be implicit in the programming language constructs and all synchronization should be limited in scope. It is our experience that requiring users to build mutual exclusion out of synchronization mechanisms, e.g. locks, often leads to incorrect programs. Further, we have noted that reducing the scope in which synchronization can be used, by encapsulating it as part of programming language constructs, further reduces errors in concurrent programs.
- Both synchronous and asynchronous communication are needed. However, we believe that the best way to support this is to provide synchronous communication as the fundamental mechanism; asynchronous mechanisms, such as buffering or futures [9], can then be built when that is appropriate. Building synchronous communication out of asynchronous mechanisms requires a protocol for the caller to subsequently detect completion. This is error prone because the caller may not obey the protocol (e.g. never retrieve a result). Further, asynchronous requests require the creation of implicit queues of outstanding requests, each of which must contain a copy of the arguments of the request. This creates a storage management problem because different requests require different amounts of storage in the queue. We believe asynchronous communication is too complicated a mechanism to be hidden in the system.
- An object that is accessed concurrently must have some control over which requester it services next. There are two distinct approaches: control can be based on the kind of request, for example, selecting a requester from the set formed by calls to a particular entry point; or control can be based on the identity of the requester. In the former case, it must be possible to give priorities to the sets of requesters. This is essential for high priority requests, such as a time out or a termination request. (This is to be differentiated from giving priority to elements within a set or execution priority.) In the latter case, selection control is very precise as the next request must only come from the specified requester. Currently, we see a need for only the former case because we believe that the need for the latter case is small.

- There must be flexibility in the order that requests are completed. This means that a task can accept a request and subsequently postpone it for an unspecified time, while continuing to accept new requests. Without this ability, certain kinds of concurrency problems are quite difficult to implement, e.g. disk scheduling, and the amount of concurrency is inhibited as tasks are needlessly blocked [10].

We have satisfied all of these requirements in μC++.

ELEMENTARY EXECUTION PROPERTIES

We have developed extensions to the object concept based on the following execution properties:

thread – is execution of code that occurs independently of other execution; a thread defines sequential execution. Conceptually, a thread is a virtual processor whose function is to advance execution by changing execution state. Multiple threads provide concurrent execution. A programming language must provide constructs that permit the creation of new threads and specify how threads are used to accomplish computation. Further, there must be programming language constructs whose execution causes threads to block and subsequently be made ready for execution. A thread is either blocked or running or ready. A thread is **blocked** when it is waiting for some event to occur. A thread is **running** when it is executing on an actual processor. A thread is **ready** when it is eligible for execution but not being executed.

execution-state – An execution-state is the state information needed to permit concurrent execution, even if it is not used for concurrent execution. An execution-state is either **active** or **inactive**, depending on whether or not it is currently being used by a thread. In practice, an execution-state consists of the data items created by an object, including its local data, local block and routine activations, and a current execution location, which is initialized to a starting point. The local block and routine activations are often maintained in a contiguous stack, which constitutes the bulk of an execution-state and is dynamic in size, and is the area where the local variables and execution location are preserved when an execution-state is inactive. A programming language knows what constitutes an execution-state, and therefore, execution-state is an elementary property of the semantics of a language. An execution-state is related to the notion of a process continuation [11]. When a thread transfers from one execution-state to another, it is called a **context switch**.

mutual exclusion – is the mechanism that permits an action to be performed on a resource without interruption by other actions on the resource. In a concurrent system, mutual exclusion is required to guarantee consistent generation of results, and cannot be trivially or efficiently implemented without appropriate programming language constructs.

The first two properties seem to be fundamental and not expressible in terms of simpler properties; they represent the minimum needed to perform execution. The last, while expressible in terms of simpler concepts, can only be done by algorithms that are error-prone and inefficient, e.g. Dekker-like algorithms, and therefore we believe that mutual exclusion must be provided as an elementary execution property.

A programming language designer could attempt to provide these 3 execution properties as basic abstractions in a programming language [12], allowing users to construct higher-level constructs from them. However, some combinations might be inappropriate or potentially dangerous. Therefore, we will examine all combinations, analyzing which combinations make sense and are appropriate as higher-level programming language constructs. What is interesting is that enumerating all combination of these elementary execution properties produces many existing high-level abstractions and suggests new ones that we believe merit further examination.

The three execution properties are properties of objects. Therefore, an object may or may not have a thread, may or may not have an execution-state, and may or may not have mutual exclusion. Different combinations of these three properties produce different kinds of objects. If an object has mutual exclusion, this means that execution of certain member routines are mutually exclusive of one another. Such a member routine is called a **mutex member**. In the situation where an object does not have the minimum properties required for execution, i.e. thread and execution-state, those of its user (caller) are used.

Table 1 shows the different abstractions possible when an object possesses different execution properties. Case 1 is objects, such as routines or class-objects that have none of the execution properties. In this case, the caller's thread and execution-state are used to perform the execution. Since this kind of object provides no mutual exclusion, it is normally accessed only by a single task. If such an object is accessed by several tasks, extreme care must be exercised to assure correct execution. Case 2 is like Case 1 but deals with the concurrent access problem by implicitly ensuring mutual exclusion for the duration of each computation by a member routine. This abstraction is a monitor [13]. Case 3 is an object that has its own execution-state but no thread. Such an object uses its caller's thread to advance its own execution-state and usually, but not always, returns the thread back to the caller. This abstraction is a coroutine [14]. Case 4 is like Case 3 but deals with the concurrent access problem by implicitly ensuring mutual exclusion; we have adopted the name coroutine-monitor for it. Cases 5 and 6 are objects with a thread but no execution-state. Both cases are rejected because the thread cannot be used to provide additional concurrency. First, the object's thread cannot execute on its own since it does not have an execution-state, so it cannot perform any independent actions. Second, if the caller's execution-state is used, assuming the caller's thread can be blocked to ensure mutual exclusion of the execution-state, the effect is to have two threads successively executing portions of a single computation, which does not seem useful. Case 7 is an object that has its own thread and execution-state. Because it has both a thread and execution-state it is capable of executing on its own; however, it lacks mutual exclusion. Without mutual exclusion, access to the object's data is unsafe; therefore, servicing of requests would, in general, require explicit locking, which violates a design requirement. Further, there is no performance advantage over case 8. For these reasons, we have rejected this case. Case 8 is like Case 7 but deals with the concurrent access problem by implicitly ensuring mutual exclusion.

object properties		object's member routine properties	
thread	execution-state	no implicit mutual exclusion	implicit mutual exclusion
no	no	1 class-object	2 monitor
no	yes	3 coroutine	4 coroutine-monitor
yes	no	5 (rejected)	6 (rejected)
yes	yes	7 (rejected)	8 task

Table 1: Fundamental Abstractions

The abstractions suggested by this categorization come from fundamental properties of execution and not ad hoc decisions of a programming language designer. While it is possible to simplify the programming language design by only supporting the task abstraction [15], which provides all the elementary execution properties, this would unnecessarily complicate and make inefficient solutions to certain problems. As will be shown, each of the non-rejected abstractions produced by this categorization has a particular set of problems that it can solve, and therefore, each has a place in the programming language. If one of these abstractions is not present, a programmer may be forced to contrive a solution for some problems that violates abstraction or is inefficient.

EXTENDING C++

Operations in μC++ are expressed explicitly, i.e. the abstractions derived from the elementary properties are used to structure a program into a set of objects that interact, possibly concurrently, to complete a computation. This is to be distinguished from implicit schemes such as those that attempt to *discover* concurrency in an otherwise sequential program, for example, by parallelizing loops and access to data structures. While both schemes are complementary, and hence, can appear together in a single programming language, we believe that implicit schemes are limited in their capacity to *discover* concurrency, and therefore, the explicit scheme is essential. Currently, μC++ only supports the explicit approach, but nothing in its design precludes implicit approaches.

The abstractions in Table 1 are expressed in μC++ using two new type specifiers, uCoroutine and uTask, which are extensions of the class construct, and hence, define new types. In this paper, the types defined by the class construct and the new constructs are called **class types**, **monitor types**, **coroutine types**, **coroutine-monitor types** and **task types**, respectively. The terms **class-object**, **monitor**, **coroutine**, **coroutine-monitor** and **task** refer to the objects created from such types. The term **object** is the generic term for any instance created from any type. All objects can be declared externally, in a block, or using the new operator. Two new type qualifiers, uMutex and uNoMutex, are also introduced to specify the presence or absence of mutual exclusion on the member routines of a type (see Table 2). The default qualification values have been chosen based on the expected frequency of use of the new types. Several new statements are added to the language: uSuspend, uResume, uCoDie, uAccept, uWait, uSignal and uDie. Each is used to affect control in objects created by the new types. (The prefix "u" followed by a capital letter for the new keywords avoids current and future conflicts with UNIX routine names, e.g. accept, wait, signal, and C++ library names, e.g. task.) Appendix A shows how the extensions are added to the C++ grammar.

object properties		object's member routine properties	
thread	execution-state	no implicit mutual exclusion	implicit mutual exclusion
no	no	[uNoMutex]† class	uMutex class
no	yes	[uNoMutex] uCoroutine	uMutex uCoroutine
yes	yes	N/A	[uMutex] uTask

† [ ] implies default qualification if not specified

Table 2: New Type Specifiers

COROUTINE

A coroutine is an object with its own execution-state so its execution can be suspended and resumed. Execution of a coroutine is suspended as control leaves it, only to carry on from that point when control returns at some later time. This means that a coroutine is not restarted at the beginning on each activation and that its local variables are preserved. Hence, a coroutine solves the class of problems associated with finite-state machines and push-down automata, which are logically characterized by the ability to retain state between invocations. In contrast, a subroutine or member routine always executes to completion before returning so that its local variables only persist for a particular invocation. A coroutine executes serially, and hence there is no concurrency implied by the coroutine construct. However, the ability of a coroutine to suspend its execution-state and later have it resumed is the precursor to true tasks but without concurrency problems; hence, a coroutine is also useful to have in a programming language for teaching purposes because it allows incremental development of these properties.

A coroutine type has all the properties of a class. The general form of the coroutine type is the following:

```
[uNoMutex] uCoroutine  coroutine-name {
  private:
    ...          // these members are not visible externally
  protected:
    ...          // these members are visible to descendants
    void main();      // starting member
  public:
    ...          // these members are visible externally
};
```

The coroutine type has one distinguished member, named main. Instead of allowing direct interaction with main, we have chosen to make its visibility private or protected; therefore, a coroutine can only be activated indirectly by one of the coroutine's member routines. A user interacts with a coroutine indirectly through its member routines. This allows a coroutine type to have multiple public member routines to service different kinds of requests that are statically type checked. main cannot have parameters, but the same effect can be accomplished indirectly by passing arguments to the constructor for the coroutine and storing these values in local variables, which can be referenced by main.

A coroutine can suspend its execution at any point by activating another coroutine. This can be done in two ways. First, a coroutine can implicitly reactivate the coroutine that previously activated it. Second, a coroutine can explicitly invoke a member of another coroutine, which causes activation of the coroutine that it is a member of. Hence, two different styles of coroutine are possible, based on whether implicit activation is used. A **semi-coroutine** always activates the member routine that activated it; a **full coroutine** calls member routines in other coroutines that cause execution of that coroutine to be activated.

Coroutines can be simulated without using a separate execution-state, e.g. using a class, but this is difficult and error-prone for more than a small number of activation points. All data needed between activations must be local to the class and the coroutine structure must be written as a series of cases, each ending by recording the next case that will be executed on re-entry. Simulating a coroutine with a subroutine requires retaining data in variables with global scope or variables with static storage-class between invocations. However, retaining state in these ways violates the principle of abstraction and does not generalize to multiple instances, since there is only one copy of the storage in both cases. Simulating a coroutine with a task, which also has an execution-state, is non-trivial because the organizational structure of a coroutine and task are different. Further, simulating full coroutines that form a cyclic call-graph is not possible with tasks because of a task's mutual exclusion, which would cause deadlock. Finally, a task is inefficient for this purpose because of the higher cost of switching both a thread and execution-state as opposed to just an execution-state. In our implementation, the cost of communication with a coroutine is, in general, less than half the cost of communication with a task, unless the communication is dominated by transferring data (see benchmark timings in Appendix D).

Coroutine Creation and Destruction

A coroutine is the same as a class-object with respect to creation and destruction, as in:

```
uCoroutine C {
    void main() ...
  public:
    void r( ... ) ...
};
C *cp;                  // pointer to a C coroutine
{ // start a new block
    C c, ca[3];         // local creation
    cp = new C;         // dynamic creation
    ...
    c.r( ... );         // call a member routine that activates the coroutine
    ca[1].r( ... );     // call a member routine that activates the coroutine
    cp->r( ... );       // call a member routine that activates the coroutine
    ...
} // c, ca[0], ca[1] and ca[2] are destroyed
...
delete cp;              // cp's instance is destroyed
```

When a coroutine is created the following occurs. The appropriate coroutine constructor and any base-class constructors are executed in the normal order. The stack component of the coroutine's execution-state is then created and the starting point (activation point) is initialized to the coroutine's main routine; however, the main routine does not start execution until the coroutine is activated by one of its member routines. The location of a coroutine's variables—in the coroutine's local data area or in routine main—depends on whether the variables must be accessed by member routines other than main. Once main is activated, it executes until it activates another coroutine or terminates. The coroutine's point of last activation may be outside of the main routine because main may have called another routine; the routine called could be local to the coroutine or in another coroutine.

A coroutine terminates when its main routine terminates or when the statement uCoDie is executed. uCoDie allows a coroutine to be terminated in a routine other than main. When a coroutine terminates, it activates the coroutine or task that caused main to start execution. This choice was made because the start sequence is a tree, i.e. there are no cycles. A thread can move in a cycle among a group of coroutines but termination always proceeds back along the branches of the starting tree. This choice for termination does impose certain requirements on the starting order of coroutines, but it is essential to ensure that cycles can be broken at termination. An attempt to communicate with a terminated coroutine is an error.

Like a routine or class, a coroutine can access all the external variables of a C++ program and the heap area. Also, any static member variables declared within a coroutine are shared among all instances of that coroutine type. If a coroutine makes global references or has static variables and is instantiated by different tasks, there is the general problem of concurrent access to such shared variables.

Inherited Members

Each coroutine type, if not derived from some other coroutine type, is implicitly derived from the coroutine type uBaseCoroutine, as in:

```
uCoroutine  coroutine-name : public uBaseCoroutine {
    . . .
};
```

where the interface for the base class uBaseCoroutine is as follows:

```
uCoroutine uBaseCoroutine {
  protected:
    void uSaveFloatRegs();
  public:
    uBaseCoroutine();
    uBaseCoroutine( int stackSize );
    void uVerify();
};
```

The protected member routine uSaveFloatRegs causes the additional saving of the floating point registers during a context switch for a coroutine. In most systems, e.g. UNIX, the entire state of the actual processor is saved during a context switch because there is no way to determine if a particular object is using only a subset of the actual processor state. All objects use the fixed-point registers, while only some use the floating-point registers. Because there is a significant execution cost in saving and restoring the floating-point registers, we have decided not to automatically save them. Hence, in μC++ the fixed-point registers are always saved during a context switch, but it may or may not be necessary to save the floating-point registers. If a coroutine or task performs floating-point operations, it must invoke uSaveFloatRegs immediately after starting execution. From that point on, both the fixed-point and floating-point registers are saved during a context switch.

The overloaded constructor routine uBaseCoroutine has the following forms:

uBaseCoroutine() – creates the coroutine with the stack size for its execution-state specified by the current cluster's default stack size, a machine dependent value no less than 4000 bytes. (A cluster is a collection of tasks and virtual processors that execute those tasks and is created from the class-type uCluster. The purpose of a cluster is to control the amount of parallelism that is possible among concurrent tasks on multiprocessor computers and is described in detail in Reference [16].)

uBaseCoroutine(int stackSize) – creates the coroutine on the current cluster with the specified stack size (in bytes).

A coroutine type can be designed to allow declarations to specify the size of the stack by doing the following:

```
uCoroutine C {
  public:
    C() : uBaseCoroutine( 8192 ) {};        // default 8K stack
    C( int i ) : uBaseCoroutine(i) {};      // user specified stack size
    . . .
};

C x, y( 16384 );      // x has an 8K stack, y has a 16K stack
```

The member routine uVerify checks whether the current coroutine has overflowed its stack. If it has, the program terminates. It is suggested that a call to uVerify be included after each set of declarations, as in the following example:

```
void main() {
    . . .               // declarations
    uVerify();          // check for stack overflow
    . . .               // code
}
```

Thus, after a coroutine has allocated its local variables, a verification is made that the stack was large enough to contain them. In the future, we plan to automatically insert calls to uVerify to ensure that any stack violation will be caught.

The free routine:

```
uBaseCoroutine &uThisCoroutine();
```

is used to determine the identity of the current coroutine. Because it returns a reference to the base coroutine type, uBaseCoroutine, this reference can only be used to access the public routines of type uBaseCoroutine. For example, a free routine can check whether the allocation of its local variables has overflowed the stack of a coroutine that called it by performing the following:

```
int FreeRtn( ... ) {
    ...                         // declarations
    uThisCoroutine().uVerify();     // check for stack overflow
    ...                         // code
}
```

Coroutine Control and Communication

Control flow among coroutines is specified by the uResume and uSuspend statements. The uResume statement is used only in the member routines; it always activates the coroutine in which it is specified, and consequently, causes the caller of the member routine to become inactive. The uSuspend statement is used only within the coroutine body, not its member routines; it causes the coroutine to become inactive, and consequently, to activate the caller that most recently activated the coroutine. In terms of the execution properties, these statements redirect a thread to a different execution-state. The execution-state can be that of a coroutine or the current task, i.e. a task's thread can execute using its execution-state, then several coroutine execution-states, and then back to the task's execution-state. Therefore, these statements activate and deactivate execution-states, and do not block and restart threads.

A semi-coroutine is characterized by the fact that it always activates its caller, as in the producer-consumer example of Figure 1. Notice the explicit call from Prod's main routine to delivery and then the return back when delivery completes. delivery always activates its coroutine, which subsequently activates delivery. Appendix B shows a complex binary insertion sort using a semi-coroutine.

A full coroutine is characterized by the fact that it never activates its caller; instead, it activates another coroutine by invoking one of its member routines. Thus, full coroutines activate one another often in a cyclic fashion, as in the producer-consumer example of Figure 2. Notice the uResume statements in routines payment and delivery. The uResume in routine payment activates the execution-state associated with Prod::main and that execution-state continues in routine Cons::delivery. Similarly, the uResume in routine delivery activates the execution-state associated with Cons::main and that execution-state continues in Cons::main initially and subsequently in routine Prod::payment. This cyclic control flow and the termination control flow is illustrated in Figure 3.

MUTEX TYPES

A mutex type is any type that has a mutex member. Objects instantiated from mutex types have special capabilities for controlling concurrent access. When uMutex or uNoMutex qualifies a type specifier, as in:

```
uMutex class M {
  private:
    char  w( ... );
  public:
    M();
    ~M();
    int x( ... );
    float y( ... );
    void z( ... );
};
```

it defines the default form of mutual exclusion on *all* public member routines, including the constructor and destructor. Hence, public member routines x, y, z, M and ~M of monitor type M are mutex members executing mutually exclusively of one another. protected and private member routines are *always* implicitly uNoMutex, except for main in coroutines and tasks. Because the destructor of mutex object is executed mutually exclusively, the termination of a block containing a mutex object or deleting a dynamically allocated one may block if the destructor cannot be executed immediately. In our implementation, a mutex member cannot call another mutex member in the same object or the executing thread deadlocks with itself.

Mutex qualifiers may be needed for protected and private member routines in mutex types, as in:

Producer	Consumer

<pre>uCoroutine Prod {
    Cons *c;
    int N, status;
 	
    void main() {
        int p1, p2;
        // 1st resume starts here
        for (int i = 1; i <= N; i += 1) {
            . . . // generate a p1 and p2
            status = c->delivery(p1, p2);
            if (status == . . .) . . .
        } // for
        c->delivery(-1, 0);
    }; // main
  public:
    Prod(Cons *c) { Prod::c = c; };
    void start(int N) {
        Prod::N = N;
        uResume;   // restart Prod::main
    }; // start
}; // Prod</pre>
	<pre>uCoroutine Cons {
	    int p1, p2, status;
	
	    void main() {
	        // 1st resume starts here
	        while (p1 >= 0) {
	            // consume p1 and p2
	            status = . . .;
	            uSuspend; // restart Cons::delivery
	        } // while
	    }; // main
	  public:
	    int delivery(int p1, int p2) {
	        Cons::p1 = p1;
	        Cons::p2 = p2;
	        uResume; // restart Cons::main
	        return status;
	    }; // delivery
	}; // Cons
	
	int main() {
	    Cons cons; // create consumer
	    Prod prod(&cons); // create producer
	
	    prod.start(10); // start producer
	} // main</pre>

Figure 1: Semi-Coroutine Producer-Consumer Solution

```
uMutex class M {
  private:
    uMutex char w( ... );      // explicitly qualified member routine
    . . .
};
```

because another thread may need access to these member routines. For example, when a friend task calls a protected or private member routine, these calls may need to provide mutual exclusion.

For convenience, we also allow a public member of a mutex type to be explicitly qualified with uNoMutex. These routines are, in general, error-prone in concurrent situations because their lack of mutual exclusion permits concurrent updating to object variables. However, there are two situations where such a non-mutex public member are useful: first, for read-only member routines where execution speed is of critical importance; and second, to encapsulate a sequence of calls to several mutex members to establish a protocol, which ensures that a user cannot violate the protocol since it is part of the object's definition.

THREAD CONTROL CONSTRUCTS

The following constructs support waiting for the arrival of a thread to and blocking a thread within a mutex object.

Controlling External Access: the Accept Statement

A uAccept statement, similar to that in Ada [17], is provided to dynamically choose which mutex member executes. This indirectly controls which caller is accepted next, that is, the next caller to the accepted mutex member. The simple form of the uAccept statement is as follows:

Producer	Consumer

<pre>uCoroutine Prod {
    Cons *c;
    int N, money, status, receipt;
 	
    void main() {
        int p1, p2;
        // 1st resume starts here
        for (int i = 1; i <= N; i += 1) {
            . . . // generate a p1 and p2
            status = c->delivery(p1, p2);
            if (status == . . .) . . .
        } // for
        c->delivery(-1, 0);
    }; // main
  public:
    int payment(int money) {
        Prod::money = money;
        . . . // process money
        uResume; // restart prod in Cons::delivery
        return receipt;
    }; // payment
    void start(int N, Cons *c) {
        Prod::N = N;
        Prod::c = c;
        uResume;
    }; // start
}; // Prod</pre>
	<pre>uCoroutine Cons {
	    Prod *p;
	    int p1, p2, status;
	
	    void main() {
	        int money, receipt;
	        // 1st suspend starts here
	        while (p1 >= 0) {
	            // consume p1 and p2
	            status = . . .
	            receipt = p->payment(money);
	        } // while
	    }; // main
	  public:
	    Cons(Prod *p) { Cons::p = p; };
	    int delivery(int p1, int p2) {
	        Cons::p1 = p1;
	        Cons::p2 = p2;
	        uResume; // restart cons in Cons::main 1st time
	                 // and cons in Prod::payment afterwards
	        return status;
	    }; // delivery
	}; // Cons
	
	int main() {
	    Prod prod; // create producer
	    Cons cons(&prod); // create consumer
	
	    prod.start(10, &cons); // start producer
	} // main</pre>

Figure 2: Full-Coroutine Producer-Consumer Solution

```
uWhen ( conditional-expression )            // optional guard for accept
    uAccept ( mutex-member-name );
```

with the restriction that the constructor, destructor, new and delete, and uNoMutex members are excluded from being accepted. The first four member routines are excluded because these routines are essentially part of the implicit memory-management runtime support. uNoMutex members are excluded because, in this implementation, they contain no entry code that could affect the caller or acceptor. While it is possible to block the caller to and the acceptor of a uNoMutex member and then continue both the caller and acceptor when they synchronize, we do not believe that such a facility is particularly useful. The syntax for accepting a mutex operator member, such as operator =, is as follows:

```
uAccept( operator = );
```

Currently, there is no way to accept a particular overloaded member. However, we have adopted the following rule when an overloaded member name appears in a uAccept statement: calls to any member with that name are accepted. The rationale is that members with the same name should perform essentially the same function, and therefore, they all should be eligible to accept a call.

A uWhen guard is considered true if it is omitted or its *conditional-expression* evaluates to non-zero. Before the uAccept statement is executed, the guard must be true and an outstanding call to the corresponding member must exist. If the guard is true and there is no outstanding call to that member, the task is accept-blocked until a call to the appropriate member is made. If the guard is false, the program is aborted; hence, the uWhen clause can act as an assertion of correctness in the simple case.

Once a call is made to an accepted mutex member, either the acceptor's or the caller's thread continues in the object because of the mutual exclusion property. We have chosen the caller's thread while the acceptor's thread

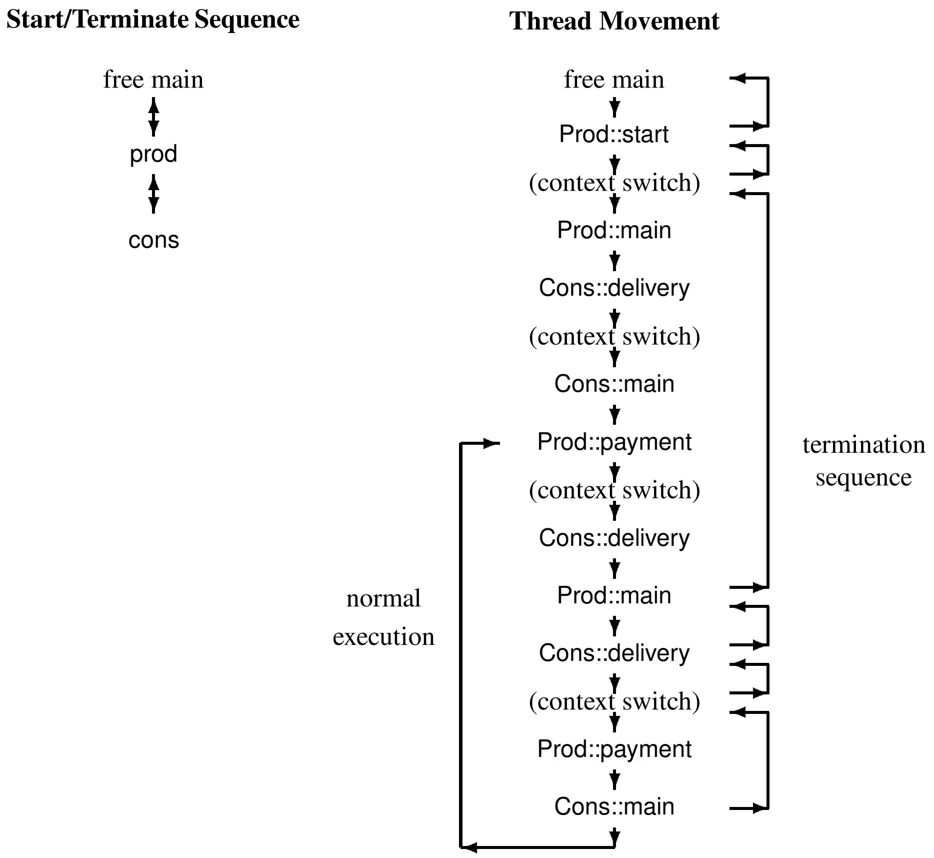

Figure 3: Cyclic Control Flow in Full Coroutine

remains blocked [18]. The accepted member is then executed like a member routine of a conventional class by the caller's thread. If the caller is expecting a return value, this value is returned using the return statement in the member routine. When the caller's thread leaves the object (or waits, as will be discussed shortly), the acceptor's thread is unblocked and has exclusive access to the object.

The extended form of the uAccept statement, similar to the Ada select statement, can be used to accept one of a group of mutex members, as in:

```
uWhen ( conditional-expression )            // optional guard for accept
    uAccept ( mutex-member-name )
        statement                           // optional statement
uOr uWhen ( conditional-expression )        // optional guard for accept
    uAccept ( mutex-member-name )
        statement                           // optional statement
...
    ...
uElse                                       // optional default clause
    statement
```

Before a uAccept clause is executed, the guard must be true and an outstanding call to the corresponding member must exist. If there are several mutex members that can be accepted, the clause nearest the beginning of the statement is executed. Hence, the order of the uAccepts indicates their relative priority for selection if there are several outstanding calls. Once the accepted call has completed, the statement after the accepting uAccept clause is executed. If there is a uElse clause and no uAccept can be executed immediately, the uElse clause is executed instead. If there is no uElse clause and the guards are all false, the program is aborted. If some guards are true and there are no outstanding calls to these members, the task is accept-blocked until a call to one of these members is made. Note that the syntax of the uAccept statement precludes the caller's argument values from being accessed in the *conditional-expression* of a uWhen. (However, this deficiency is handled by the ability of a task to postpone requests, as is discussed shortly.)

In contrast to Ada, a uAccept statement in μC++ places the code to be executed in a public member routine of the task type; thus, it is specified separately from the uAccept statement. An Ada-style accept specifies the accept body as part of the accept statement, requiring the accept statement to provide parameters and a routine body. Since we have found that having more than one accept statement per member is rather rare, our approach gives essentially the same capabilities as Ada. As well, accepting member routines also allows virtual routine redefinition, which is not possible with accept bodies. Finally, an accept statement with parameters and a routine body does not fit with the design of C++ because it is like a nested routine definition, and since routines cannot be nested in C++, there is no precedent for such a facility. It is important to note that anything that can be done in Ada-style accept statements can be done within member routines, possibly with some additional code. If members need to communicate with the block containing the uAccept statements, it can be done by leaving "memos" in the task's local variables. In cases where there would be several different Ada-style accept statements for the same entry, accept members would have to start with switching logic to determine which case applies.

Condition Variables and the Wait and Signal Statements

Sometimes it is necessary for the actions performed by a member routine to be postponed until some later time. This can be done using uCondition variables in conjunction with the uWait and uSignal statements. The type uCondition creates a queue object on which tasks can be blocked and restarted in first-in first-out order, and is defined:

```
class uCondition {
  public:
    int uEmpty();
};
```

The member routine uEmpty() returns 0 if there are threads blocked on the queue and 1 otherwise. It is *not* meaningful to read or to assign to a condition variable, or copy a condition variable (e.g. pass it as a value parameter), or use a condition variable outside of the mutex object in which it is declared.

The uWait statement blocks the current thread on a condition. Because of the mutual exclusion property, there can be pending threads from uAccept or uSignal statements (discussed next) that want to continue execution; only one can be restarted. The only sensible choice is to restart them in last-blocked first-unblocked (stack) order. If there are no pending threads, the only choice is to do an accept of all the mutex members so that some thread can subsequently enter the object and signal a waiting thread; otherwise, deadlock will occur.

The uSignal statement makes the thread on the front of the condition queue ready; if there is no thread on the condition queue, the signaller continues execution. This is different from the uAccept statement, which blocks if there are no outstanding calls to a mutex member. Once a thread is unblocked by uSignal, only one of the two threads can continue in the object because of the mutual exclusion property. We have chosen the signalled thread. When the signalled thread leaves the object or waits, the signaller's thread is automatically unblocked and it has exclusive access to the object.

The uAccept, uWait and uSignal statements can be executed by any routine of a mutex type. Even though these statements block the current task, they can be allowed in member routines because member routines are executed by their caller, not the task of which they are members. This is to be contrasted to Ada where the use of a statement like a uWait in an accept body would cause the task to deadlock.

Postponing Requests

We consider the ability to postpone a request to be essential. Postponement may occur multiple times during servicing of a request while still allowing an object to accept new requests.

In simple cases, the uWhen construct can be used to accept only requests that can be completed without postponement. However, when the selection criteria become complex, e.g. when the parameters of the request are needed to do the selection or information is needed from multiple queues, it is simpler to unconditionally accept a request and subsequently postpone it if it does not meet the selection criteria. This avoids complex selection expressions and possibly their repeated evaluation. In addition, this allows all the programming language constructs and data structures to be used in making the decision to postpone a request, instead of some fixed selection mechanism provided in the programming language, as in SR [19] and Concurrent C++ [20].

Regardless of the power of a selection facility, none can deal with the need to postpone a request after it has been accepted. In a complex concurrent system, a task may have to make requests to other tasks as part of servicing a request. Any of these further requests can indicate that the current request cannot be completed at this time and must be postponed. Thus, we believe that it is essential that a request be able to be postponed even after it is accepted so that the acceptor can make this decision while the request is being serviced. Therefore, condition variables seem essential to support this facility.

Ultimately, we want to restart tasks blocked within a terminating mutex object—on entry queues and condition variables—by raising an exception to notify them that their call failed. We are currently examining how to incorporate exceptions among tasks within the proposed C++ exception model.

MONITOR

A monitor is an object with mutual exclusion and so it cannot be accessed simultaneously by multiple threads. A monitor provides a mechanism for indirect communication among tasks and is particularly useful for managing shared resources. A monitor type has all the properties of a class. The general form of the monitor type is the following:

```
uMutex class monitor-name {
  private:
    . . .        // these members are not visible externally
  protected:
    . . .        // these members are visible to descendants
  public:
    . . .        // these members are visible externally
};
```

Monitor Creation and Destruction

A monitor is the same as a class-object with respect to creation. Because a monitor is a mutex object, the execution of its destructor waits until it can gain access to the monitor, just like the other public members of the monitor, which can delay the termination of the block containing a monitor or the deletion of a dynamically allocated monitor.

Monitor Control and Communication

uWait, uSignal, and uCondition variables provide the same functionality as for a conventional monitor [13]. In μC++, the uAccept statement can also be used to control which member(s) can be executed next. The ability to use uAccept in a monitor makes it more general than a conventional monitor because it specifies a mutex member, while uSignal specifies only a condition variable. This gives the ability to restrict which member can be called, instead of having to accept all calls and subsequently handle or block them, as for conventional monitors. Figure 4 compares a traditional style monitor using explicit condition variables to one that uses accept statements. The problem is the exchange of values (telephone numbers) between two kinds of tasks (girls and boys). (uAccept allows the removal of all condition variables in this case, but that is not always possible.)

COROUTINE-MONITOR

The coroutine-monitor is a coroutine with mutual exclusion and so it cannot be accessed simultaneously by multiple threads. A coroutine-monitor type has a combination of the properties of a coroutine and a monitor, and can be used where a combination of these properties are needed, such as a finite-state machine that is used by multiple tasks. A coroutine-monitor type has all the properties of a class. The general form of the coroutine-monitor type is the following:

Traditional Method	New Method

<pre>uMutex class DatingService {
    int GirlPhoneNo, BoyPhoneNo;
    uCondition GirlWaiting, BoyWaiting;
  public:
    int Girl(int PhoneNo) {
        if (BoyWaiting.uEmpty()) {
            uWait GirlWaiting;
            GirlPhoneNo = PhoneNo;
        } else {
            GirlPhoneNo = PhoneNo;
            uSignal BoyWaiting;
        } // if
        return BoyPhoneNo;
    }; // Girl
    int Boy(int PhoneNo) {
        if (GirlWaiting.uEmpty()) {
            uWait BoyWaiting;
            BoyPhoneNo = PhoneNo;
        } else {
            BoyPhoneNo = PhoneNo;
            uSignal GirlWaiting;
        } // if
        return GirlPhoneNo;
    }; // Boy
}; // DatingService</pre>
	<pre>uMutex class DatingService {
	    int GirlPhoneNo, BoyPhoneNo;
	  public:
	    DatingService() {
	        GirlPhoneNo = BoyPhoneNo = -1;
	    }; // DatingService
	    int Girl(int PhoneNo) {
	        GirlPhoneNo = PhoneNo;
	        if (BoyPhoneNo == -1) {
	            uAccept(Boy);
	        } // if
	        int temp = BoyPhoneNo;
	        BoyPhoneNo = -1;
	        return temp;
	    }; // Girl
	    int Boy(int PhoneNo) {
	        BoyPhoneNo = PhoneNo;
	        if (GirlPhoneNo == -1) {
	            uAccept(Girl);
	        } // if
	        int temp = GirlPhoneNo;
	        GirlPhoneNo = -1;
	        return temp;
	    }; // Boy
	}; // DatingService</pre>

Figure 4: Traditional versus New Monitor Control

```
uMutex uCoroutine coroutine-name {
  private:
    . . .           // these members are not visible externally
  protected:
    . . .           // these members are visible to descendants
    void main();       // starting member
  public:
    . . .           // these members are visible externally
};
```

Currently, we have little experience in using a coroutine-monitor, nevertheless we believe it merits further examination.

Coroutine-Monitor Creation and Destruction

A coroutine-monitor is the same as a monitor with respect to creation and destruction.

Coroutine-Monitor Control and Communication

A coroutine-monitor can make use of uSuspend, uResume, uAccept and uCondition variables, uWait and uSignal to move a thread among execution-states and to block and restart threads that enter it.

TASK

A task is an object with its own thread of control and execution-state, and whose public member routines provide mutual exclusion. A task type has all the properties of a class. The general form of the task type is the following:

```
[uMutex] uTask task-name {
  private:
     . . .          // these members are not visible externally
  protected:
     . . .          // these members are visible to descendants
     void main();      // starting member
  public:
     . . .          // these members are visible externally
};
```

The task type has one distinguished member, named main, in which the new thread starts execution. Instead of allowing direct interaction with main, we have chosen to make its visibility private or protected. A user then interacts with a task's main member indirectly through its member routines. This allows a task type to have multiple public member routines to service different kinds of requests that are statically type checked. main cannot have parameters, but the same effect can be accomplished indirectly by passing arguments to the constructor for the task and storing these values in the task's local variables, which can be referenced by main. If a C++ program is viewed as an instance of an object where a thread of control begins in the free routine main, a task is just a generalization of this notion.

Task Creation and Destruction

A task is the same as a class-object with respect to creation and destruction, as in:

```
uTask T {
     void main() . . .
  public:
     void r( . . . ) . . .
};
T *tp;                    // pointer to a T
{ // start a new block
     T t, ta[3];          // local creation
     tp = new T;          // dynamic creation
     . . .
     t.r( . . . );        // call a member routine that must be accepted
     ta[1].r( . . . );    // call a member routine that must be accepted
     tp->r( . . . );      // call a member routine that must be accepted
     . . .
} // wait for t, ta[0], ta[1] and ta[2] to terminate and then destroy
. . .
delete tp;                // wait for tp's instance to terminate and then destroy
```

When a task is created, the appropriate task constructor and any base-class constructors are executed in the normal order by the creating thread. Then a new thread of control and execution-state are created for the task, which are used to begin execution of the main routine visible by the inheritance scope rules from the task type. From this point, the creating thread executes concurrently with the new task's thread. main executes until its thread blocks or terminates.

A task terminates when its main routine terminates or when its destructor is accepted or when the statement uDie is executed. uDie allows a task to be terminated in a routine other than main. When a task terminates, so does the task's thread of control and execution-state. A task's destructor is invoked by the destroying thread when the block containing the task declaration terminates or by an explicit delete statement for a dynamically allocated task. However, the destructor must not execute before the task's thread terminates because the storage for the task is released by the destructor. Therefore, a μC++ block cannot terminate until all tasks declared in the block terminate. Deleting a task on the heap must also wait until the task being deleted has terminated. An attempt to communicate with a terminated task is an error.

While a task that creates another task is conceptually the parent and the created task its child, μC++ makes no implicit use of this relationship nor does it provide any facilities based on this relationship. Once a task is declared it has no special relationship with its declarer other than what results from the normal scope rules.

Like a coroutine, a task can access all the external variables of a C++ program and the heap area. However, because tasks execute concurrently, there is the general problem of concurrent access to such shared variables. Further, this problem may also arise with static member variables within a task that is instantiated multiple times. Therefore, it is suggested that these kinds of references be used with extreme caution.

Inherited Members

Each task type, if not derived from some other task type, is implicitly derived from the task type uBaseTask, as in:

```
uTask task-name : public uBaseTask {
    . . .
};
```

where the interface for the base class uBaseTask is as follows:

```
uTask uBaseTask : uBaseCoroutine {    // inherit from coroutine base type
  public:
    uBaseTask();
    uBaseTask( int stackSize );
    uBaseTask( uCluster &cluster );
    uBaseTask( int stackSize, uCluster &cluster );
    void uDelay( int times = 1 );
    uCluster &uMigrate( uCluster &cluster );
};
```

The protected member routine uSaveFloatRegs and the public member routine uVerify are inherited from uBaseCoroutine, and they have the same functionality.

The overloaded constructor routine uBaseTask has the following forms:

uBaseTask() – creates the task on the current cluster with the stack size for its execution-state specified by the current cluster's default stack size, a machine dependent value no less than 4000 bytes (same as uBaseCoroutine()).

uBaseTask(int stackSize) – creates the task on the current cluster with the specified stack size (in bytes) (same as uBaseCoroutine(int stackSize)).

uBaseTask(uCluster &cluster) – creates the task on the specified cluster with the stack size specified by that cluster's default stack size, a machine dependent value no less than 4000 bytes.

uBaseTask(int stackSize, uCluster &cluster) – creates the task on the specified cluster with the specified stack size (in bytes).

A task type can be designed to allow declarations to specify the cluster on which creation occurs and the size of the stack by doing the following:

```
uTask T {
  public:
    T() : uBaseTask( 8192 ) {};                    // current cluster, 8K stack
    T( int i ) : uBaseTask( i ) {};                // current cluster and user stack size
    T( uCluster &c ) : uBaseTask( c ) {};          // user cluster
    T( int i, uCluster &c) : uBaseTask(c,i) {};    // user cluster and stack size
    . . .
};
uCluster c;             // create a new cluster
T x, y( 16384 );        // x has an 8K stack, y has a 16K stack
T z( c );               // z created in cluster c with default stack size
T w( 16384, c );        // w created in cluster c and has a 16K stack
```

The public member routine routine uDelay gives up control of the cluster processor to another ready task the specified number of times. For example, the routine call uDelay(5) returns control to the μC++ kernel to schedule another task, hence immediately giving up control of the processor and ignoring the next 4 times the task is scheduled for execution. If there are no other ready tasks, the delaying task is restarted. uDelay allows a task to relinquish control when it has no current work to do or when it wants other ready tasks to execute before it performs more work. An example of the former situation is when a task is polling for an event, such as a hardware event. After the polling task has determined the event has not occurred, it can relinquish control to another ready task, e.g. uDelay(1). An example of the latter situation is when a task is creating many other tasks. The creating task may not want to create a large number of tasks before the created tasks have a chance to begin execution. (Task creation occurs so quickly that it is possible to create 30-50 tasks before pre-emption occurs.) If after the creation of several tasks the creator yields control, then some created tasks will have an opportunity to begin execution before the next group of tasks is created. This facility is not a mechanism to control the exact order of execution of tasks; pre-emptive scheduling and/or multiple processors make this impossible.

Most tasks execute on only one cluster. However, when it is necessary to do so, the member routine uMigrate is used to move a task from one cluster to another so that it can access resources that are dedicated to that cluster's processor(s).

from-cluster-reference = *task*.uMigrate(*to-cluster-reference*)

The free routine:

```
uBaseTask &uThisTask();
```

is used to determine the identity of the current task. Because it returns a reference to the base task type, uBaseTask, of the current task, this reference can only be used to access the public routines of type uBaseTask and uBaseCoroutine. For example, a free routine can delay execution of the calling task by performing the following:

```
int FreeRtn( ... ) {
    ...                    // declarations
    uThisTask().uDelay();  // delay execution
    ...                    // code
}
```

Task Control and Communication

A task can make use of uAccept and uCondition variables, uWait and uSignal to block and restart threads that enter it. Appendix C shows the archetypical disk scheduler implemented as a task that must process requests in an order other than first-in first-out to achieve efficient utilization of the disk.

INHERITANCE

While concurrency and our set of elementary execution properties do not imply any particular polymorphism/reuse mechanism, C++ supports these concepts using inheritance; therefore, this section examines how inheritance affects our constructs. C++ provides two forms of inheritance: "private" inheritance, which provides code reuse, and "public" inheritance, which provides reuse and subtyping (a promise of behavioural compatibility). (These terms must not be confused with C++ visibility terms with the same names.)

In C++ there is only one kind of type specifier, class; class definitions can inherit from one another using both single and multiple inheritance. In μC++ there are three kinds of types, class, coroutine, and task, so the situation is more complex. The trivial case where a class or coroutine or task type inherits from another class or coroutine or task, respectively, is supported in μC++. When coroutines and tasks inherit from other such types, each entity in the hierarchy may specify a main member; the main member specified in the last derived class of the hierarchy is the one that is started when a new instance is created. Clearly, there must be at least one main member specified in the hierarchy. For a task or a monitor type, new member routines that are defined by the derived class can be accepted by statements in a new main routine or in redefined virtual routines.

Having mutex types inherit from non-mutex types may be useful to generate concurrent usable types from existing non-concurrent types, for example, to define a queue that is derived from a simple queue and that can be accessed concurrently. However, there is a fundamental problem with non-virtual members in C++. To change a simple queue to a sharable queue, for example, would require a monitor to inherit from the class Queue and to redefine all of the class's member routines so that the correct mutual exclusion occurred when they are invoked. However, non-virtual routines in the Queue class might be called instead because non-virtual routines are statically bound. Consider, this attempt to create a sharable queue from a non-sharable queue:

```
class Queue {
  public:
    void insert( ... ) ...
    virtual void remove( ... ) ...
};
uMutex class MutexQueue : public Queue {
    virtual void insert( ... ) ...
    virtual void remove( ... ) ...
};
Queue *qp = new MutexQueue; // subtyping allows assignment
qp->insert( ... );          // call to a non-virtual member routine, statically bound
qp->remove( ... );          // call to a virtual member routine, dynamically bound
```

Routine Queue::insert does not provide mutual exclusion because it is a member of the class, while routines MutexQueue::insert and MutexQueue::remove do provide mutual exclusion. Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert was redefined in MutexQueue; so no mutual exclusion occurs. In contrast, the call to remove is dynamically bound, so the redefined routine in the monitor is invoked and appropriate synchronization occurs. The unexpected lack of mutual exclusion would cause many errors. In object-oriented programming languages that have only virtual member routines, this is not a problem. The problem does not occur with private inheritance because no subtype relationship is created and hence the assignment to qp would be invalid.

Currently, we do not support inheritance among different kinds of types. While there are some implementation difficulties with certain combinations and potential problems with non-virtual routines, the main reason is a fundamental one. Types are written as a class or a coroutine or a task possibly with mutual exclusion, and we do not believe that the coding styles used in each can be arbitrarily mixed. For example, an object produced by a task that inherits from a class can be passed to a routine expecting instances of the class and the routine might call one of the object's member routines that inadvertently blocks the current thread indefinitely. While this could happen in general, we believe there is a significantly greater chance if users casually combine types of different kinds. We plan to study this design decision with the expectation of relaxing it in the future.

IMPLEMENTATION AND PERFORMANCE

Our current implementation is a translator that reads a program containing the language extensions and translates each extension into one or more C++ statements, which are then compiled by an appropriate C++ compiler and linked with a concurrency runtime library, called the μC++ kernel. Because μC++ is only a translator and not a compiler, some restrictions apply that would be unnecessary if the extensions were actually part of a C++ programming language extension. For example, some runtime member routines are publicly visible when they should not be; therefore, μC++ programs should not contain variable names that start with a "u" followed by a capital letter. Similar, but less extensive translators have been built: Concurrent C++ [20] and MC [21]. All programming examples in this paper compile and execute using the μC++ translator and concurrency kernel.

The μC++ kernel is a library of classes and routines that provide low-level light-weight concurrency support on uniprocessor and multiprocessor computers running the UNIX* operating system. The μC++ kernel does not call the UNIX kernel to perform a context switch or to schedule tasks, and it uses shared memory for communication. As a result, performance for execution of and communication among large numbers of tasks is significantly increased over UNIX processes. The maximum number of tasks that can exist is restricted only by the amount of memory available in a program. The minimum stack size for an execution-state is machine dependent, but is as small as 256 bytes. The storage management of all μC++ objects, the scheduling of tasks on virtual processors within clusters, and the pre-emptive round-robin scheduling to interleave task execution is performed by the μC++ kernel.

Unfortunately, the μC++ kernel is not yet fully integrated with all the functionality needed by the μC++ constructs. An example problem is that a task cannot be declared in the external area because the μC++ kernel may not have started execution before the first task is initialized. As well, μC++ allows at most 32 mutex members because a 32 bit mask is used to test for accepted member routines. This approach does not extend to support multiple inheritance. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [22, 23], nor do we believe that the performance degradation required to support multiple inheritance is acceptable.

*UNIX is a registered trademark of AT&T Bell Laboratories

Benchmark times for several of the key operations in μC++ are given in Appendix D. It takes only one microsecond to create a class in a block because there is no initialization of any class variables; the one microsecond is the time to allocate the necessary storage on the stack and execute an iteration of the test loop. It takes more time to create a monitor because the mutex data structure is initialized and there is also an initialization cost for each mutex routine. Notice that the cost for operations such as accepting a call or resuming/suspending or signaling/waiting are the same in either uniprocessor or multiprocessor because they are independent of the kind of object. Certain multiprocessor timings are higher than their uniprocessor counterparts because of the extra synchronization code necessary on multiprocessor machines.

COMPARISON WITH OTHER WORK

The comparison section is divided into a discussion of concurrency libraries that have been built using C++ and some programming languages that have attempted to solve problems raised in this paper.

Concurrency Libraries

Initially, we attempted to add the new types and statements by creating a library of class definitions that were used through inheritance and preprocessor macros. This approach has been used to provide coroutine facilities [24, 25] and simple parallel facilities [26, 12]. For example, the library approach involves defining an abstract class, Task, which implements the task abstraction. New task types are created by inheritance from Task, and tasks are instances of these types.

Task creation must be arranged so that the task body does not start execution until all of the task's initialization code has finished. One approach requires the task body to be placed at the end of the new class's constructor, with code to start a new thread in Task::Task(). One thread then continues normally, returning from Task::Task() to complete execution of the constructors, while the other thread returns directly to the point where the task was declared. This is accomplished in the library approach by having one thread "diddle" with the stack to find the return address of the constructor called at the declaration. However, this scheme prevents further inheritance; it is impossible to derive a type from a task type if the new type requires a constructor, since the new constructor would be executed only *after* the parent constructor containing the task body. It also seems impossible to write stack diddling code which causes one thread to return directly to the declaration point if the exact number of levels of inheritance is not known. We tried to implement another approach that did not rely on stack diddling while still allowing inheritance and found it was impossible because a constructor cannot determine if it is the last constructor in an inheritance chain. Therefore, it is not possible to determine when all initialization is completed so that the new thread can be started.

PRESTO solved this problem by providing a start() member routine in class Task, which must be called after the creation of a task. Task::Task() would set up the new thread, but start() would set it running. However, this two-step initialization introduces a new user responsibility: to invoke start before invoking any member routines or accessing any member variables.

A similar two-thread problem occurs during deletion when the destructors are called. The destructor of a task can be invoked while the task body is executing, but clean-up code must not execute until the task body has terminated. Therefore, the code needed to wait for a thread's termination cannot simply be placed in Task::\verb+~+Task(), because it would be executed after all the derived class destructors have executed. Task designers could be required to put the termination code in the new task type's destructor, but that would prevent further inheritance. Task could provide a finish() routine, analogous to start(), which must be called before task deletion, but that is error-prone because a user may fail to call finish appropriately, for example, before the end of a block containing a local task.

Communication among tasks also presents difficulties. In library-based schemes, it is often done via message queues. However, a single queue per task is inadequate; the queue's message type inevitably becomes a union of several "real" message types, and static type checking is compromised. (One could use inheritance from a Message class, instead of a union, but the task would still have to perform type tests on messages before accessing them.) If multiple queues are used, some analogue of the Ada select statement is needed to allow a task to block on more than one queue. There is also no way to statically state that a queue is owned by one task; this facility is necessary to preclude multiple tasks from selecting from potentially overlapping sets of queues. This would be expensive since the addition or removal of a message to/from a queue would have to be an atomic operation across all queues. Finally, message queues are best defined as generic data structures, but C++ does not yet support generics.

If the more natural routine-call mechanism is to be used for communication among tasks, each public member routine would have to have special code at the start and possibly at the exits of each public member, which the programmer would have to provide. Other object-oriented programming languages that support inheritance of routines, such as LOGLAN'88 [27] and Beta, [28] or wrapper routines, as in GNU C++ [29], might be able to provide automatically any special member code. Further, we could not find any convenient way to provide an Ada-like select statement without extending the language.

In the end, we found the library approach to be unsatisfactory. We decided that language extensions would better suit our goals by providing more flexible and consistent primitives, and static checking. It is also likely that language extensions could provide greater efficiency than a set of library routines.

Concurrent Programming Languages

There are a large number of concurrency designs in an equally large number of programming languages. For this comparison, we have selected only programming languages that provide statically type-safe communication. In all the languages examined, requests can be postponed by establishing a protocol between client and server, which may require creation of new threads for or implicit buffering of each request. However, as stated in the design requirements, a protocol can violate the server's abstraction if the client does not adhere to it; therefore, solutions involving a protocol are deemed inadequate.

Ada

Many of the Ada constructs have been discussed through contrasts with μC++ constructs in previous parts of the paper. Briefly, Ada supports concurrency through an object-based construct called a task, which has multiple entry points. It provides an accept statement for accepting calls to a particular entry and a select statement for accepting calls to one of a number of entries. A task type extends the task construct by allowing instantiation of multiple instances of the type making it almost identical to a class. However, a task type differs from a traditional class type because the body of an entry is specified with the accept statement, rather than with the definition of the entry itself.

There are four major problems with the Ada approach to concurrency:

1. There is no direct mechanism to return values from an entry.
2. An accept can only appear in the body of a task.
3. The when clause of a select cannot access the arguments of the accept call.
4. Only a static number of requests can be postponed while continuing to accept new requests, which is further restricted by not being able to recursively accept a particular entry.

The first restriction is a syntactic annoyance as it is possible to return values through the argument-parameter mechanism. The second restriction is more serious as it affects the ability to modularize a program; nevertheless, it is still possible to write programs, although awkwardly in some cases. The third restriction makes it impossible to use the state of the request to determine if a request should be accepted. The fourth restriction is very serious as it makes programs that sensibly need this capability complicated or impossible to implement. Solutions presented to overcome the third and fourth restriction, e.g. entry families, are not general [30, section 13.2.11].

Our design does not have problems 1, 2 and 4, and problem 3 is handled indirectly by the solution to 4, which is the ability to receive a request, examine its parameter values, and then postpone it.

BNR Pascal

BNR Pascal [31] provides a task similar to our process construct with type QUEUE, and DEFER and REENTER statements, which are like our uCondition, uWait and uSignal, respectively. Postponing is done with DEFER(*queue-name*), which postpones the current request on the specified queue. Re-acceptance is done with REENTER(*queue-name*), which restarts the process at the head of the queue (if any) at the beginning of the entry routine it initially called, instead of continuing from the point of its suspension. Unfortunately, this scheme causes code at the beginning of a deferred member to be executed multiple times, which can result in complex switching logic at the start of entry routines to determine whether or not execution has resulted from a REENTER, and if so, why it was deferred.

SR

SR [19] provides concurrent objects that can have both concurrent member routines and entry members with Ada style accept statements. The way in which a member is invoked (by a call or send) indicates whether or not the caller blocks until the called member is finished. (A member declaration can state that it can only be called or only sent to.) The four combinations of call type and member type give SR remote procedure call, Mesa-style process emission, Ada-style rendezvous and non-blocking message passing.

Our approach and the SR approach can build identical concurrency abstractions. The main difference is in the way these abstractions are accessed by a user. SR's send provides a form of asynchronous communication (called semisynchronous), while our system would require a user to explicitly buffer requests or create a courier or future task to perform a call and wait for its completion (and possibly a result). As stated in the design requirements, we do not believe asynchronous communication should be a primitive capability of the programming language because it is too complex a mechanism to be hidden, and SR's semisynchronous communication requires automatic buffering.

SR does not provide a mechanism to postpone an accepted request. It does provide facilities to selectively accept requests, and this capability allows access to the arguments of an entry call.

As an aside, SR also provides a by clause to search a particular queue of pending requests. The by clause cannot be used to selectively decide to accept an entry, only to choose among available requests after a decision to accept an entry has been made. However, there is a class of problems that cannot be handled using a by clause [32]. These problems are characterized by a selection criteria that involves information on two or more entry queues. For example, tasks G and B each call different entry routines of a task and the acceptance criteria involves testing data from each others arguments, as in, a G task with argument X must wait until a B task with an argument X appears and vice versa.

Concurrent C++

Our work is similar to that done in Concurrent C++ [20]. However, our design is significantly more extensive in facilities provided and more general in the ability of a task to service requests in arbitrary order. As well, our design is a cleaner integration of concurrency into C++ because we did not have to support concurrency within C [33] using the same mechanism. Concurrent C++ provides the same constructs for deferring requests and searching queues as SR and, like SR, it cannot postpone an accepted request.

CONCLUSIONS

During the course of this work we have convinced ourselves of the following points:

- Three fundamental execution properties, thread, execution-state, and mutual exclusion, when combined in different ways lead to many existing programming language abstractions and several new and interesting ones. These execution properties are fundamental to the execution of all programs on von Neumann machines. This systematic approach to justifying what abstractions to introduce in a programming language is extremely important. It shows that programming language design does not have to be based solely on designer biases, but can be derived from a small set of basic primitives. While we are not yet convinced that all the abstractions presented by this particular set of elementary properties will be useful in every day programming, at this time we cannot eliminate any of them. Only concrete experience with these abstractions will provide the necessary information to make these decisions.

- Statically type-safe direct communication is essential for reliable efficient concurrency and it can be supported by standard argument-parameter communication in routine calls, independent of any other aspects such as polymorphism/reuse.

- Controlling which request is accepted next and servicing requests in arbitrary order can be supported in a statically type-safe way. This includes the ability to postpone a request after it is accepted, possibly multiple times during its servicing, while continuing to accept new requests.

- All mutual exclusion can be made implicit by incorporating it as a special property of calls to certain kinds of objects. Further, the scope of synchronization can be restricted to within objects by restricting the scope of condition variables.

Finally, it is possible to embed all of these facilities in the language C++ in a reasonably graceful and efficient way.

ACKNOWLEDGMENTS

We would like to thank Mike Coffin for reading and commenting on this work.

APPENDIX A μC++ GRAMMAR

The grammar for μC++ is an extension of the grammar for C++ given in Reference [1, Chapter 17]. The ellipsis in the following rules represent the productions elided from the C++ grammar.

type-specifier:
  . . .
  uMutex
  uNoMutex
class-key:
  . . .
  uCoroutine
  uTask
statement:
  . . .
  uDie ;
  uCoDie ;
  uWait *expression* ;
  uSignal *expression* ;
  uSuspend ;
  uResume ;
  accept-statement
when-clause:
  uWhen (*expression*)
accept-statement:
  when-clause$_{opt}$ uAccept (*dname*) *statement*
  when-clause$_{opt}$ uAccept (*dname*) *statement* uOr *accept-statement*
  when-clause$_{opt}$ uAccept (*dname*) *statement* uElse *statement*

APPENDIX B COROUTINE BINARY INSERTION SORT

The coroutine BinarySort inputs positive integer values to be sorted and sorts them using the binary insertion sort method. For each integer in the set to be sorted, BinarySort is restarted with the integer as the argument. The end of the set of integers to be sorted is signaled with the value -1. When the coroutine receives a value of -1, it stops sorting and prepares to return the sorted integers one at a time. To retrieve the sorted integers, BinarySort is restarted once for each integer in the sorted set. Each restart returns as its result the next integer of the sorted set. The last value returned by BinarySort is -1, which denotes the end of the sorted set, and then BinarySort terminates.

If the set of integers contains more than one value, BinarySort sorts them by creating two more instances of BinarySort, and having each of them sort some of the integers. Each of the two new coroutines may eventually have to create two more coroutines in turn. The result is a binary tree of coroutines.

```
uCoroutine BinarySort {
    int in, out;
    void main();
  public:
    void input( int val ) {
        in = val;
        uResume;
    }; // input
    int output() {
        uResume;
        return out;
```

```
    }; // input
}; // BinarySort

void BinarySort::main() {
    int pivot;

    pivot = in;                                 // first value is the pivot value
    if ( pivot == -1 ) {                        // no data values
        uSuspend;                               // acknowledge end of input
        out = -1;
        return;                                 // terminate output
    } // if

    BinarySort less, greater;                   // create siblings

    for ( ;; ) {
        uSuspend;                               // get more input
      if ( in == -1 ) break;
        if ( in <= pivot ) {                    // direct value along appropriate branch
            less.input( in );
        } else {
            greater.input( in );
        } // if
    } // for

    less.input( -1 );                           // terminate input
    greater.input( -1 );                        // terminate input
    uSuspend;                                   // acknowledge end of input

    // return sorted values

    for ( ;; ) {
        out = less.output();                    // retrieve the smaller values
      if ( out == -1 ) break;                   // no more smaller values ?
        uSuspend;                               // return smaller values
    } // for

    out = pivot;
    uSuspend;                                   // return the pivot

    for ( ;; ) {
        out = greater.output();                 // retrieve the larger values
      if ( out == -1 ) break;                   // no more larger values ?
        uSuspend;                               // return larger values
    } // for

    out = -1;
    return;                                     // terminate output
} // BinarySort::main
```

APPENDIX C DISK SCHEDULER

The following example illustrates a fully implemented disk scheduler using μC++. The disk scheduling algorithm is the elevator algorithm, which services all the requests in one direction and then reverses direction. A linked list is used to store incoming requests while the disk is busy servicing a particular request. The nodes of the list are stored on the stack of the calling tasks so that postponing does not consume resources. The list is maintained in sorted order by cylinder number and there is a pointer which scans backward and forward through the list. New requests can be added both before and after the scan pointer while the disk is busy. If new requests are added before the scan pointer in the direction of travel, they are serviced on that scan.

To prevent deadlocks between the disk and disk scheduler, the disk calls the scheduler to get the next request that it services. This call does two things: it passes to the scheduler the status of the just completed disk request, which is then returned from scheduler to disk user, and it returns the information for the next disk operation. When a user's request is accepted, the parameter values from the request are copied to local variables in the scheduler and then the user waits on the condition queue CurrentRequest. Hence, the scheduler only has a single copy of the request that is currently being serviced by the disk. The cost is the restarting of the user to retrieve the values from their stack, which is small in a shared memory environment.

```
typedef char Buffer[50];                                    // dummy data buffer

const int NoOfCylinders = 100;
enum IOStatus { IO_COMPLETE, IO_ERROR };

class IORequest {
  public:
    int track;
    int sector;
    Buffer *bufadr;
}; // IORequest

class WaitingRequest : public Sequable {                   // element for a waiting request list
  public:
    int track;
    uCondition req;
    WaitingRequest( int track ) { WaitingRequest::track = track; }
}; // WaitingRequest

declare(Sequence, WaitingRequest);                          // generic doubly linked list

class Elevator : public Sequence(WaitingRequest) {
    . . .                                                   // make doubly linked list into an ordered list
    }; // Elevator

uTask DiskScheduler {
    Elevator PendingClients;                                // ordered list of client requests
    boolean DiskInUse;
    WaitingRequest *ActiveClient;
    IOStatus ActiveStatus;
    IORequest ActiveRequest;

    uCondition CurrentRequest, DiskWaiting;
    void main();
  public:
    IORequest WorkRequest( IOStatus );
    IOStatus DiskRequest( IORequest & );
    void Die();
}; // DiskScheduler

void DiskScheduler::main() {
    Disk disk( *this );                                     // start the disk

    DiskInUse = TRUE;
    ActiveClient = NULL;

    for ( ;; ) {
        uAccept( Die ) {                                    // request from system
            break;
        } uOr uAccept( WorkRequest ) {                      // request from disk
        } uOr uAccept( DiskRequest ) {                      // request from clients
        } // uAccept
    } // for
    ActiveRequest.track = -1;                               // terminate the disk
    uSignal DiskWaiting;
} // DiskScheduler::main

IOStatus DiskScheduler::DiskRequest( IORequest &req ) {
    WaitingRequest np( req.track );                         // preallocate waiting list element

    if ( !DiskInUse ) {                                     // disk free ?
        DiskInUse = TRUE;
        ActiveClient = &np;
```

```
        ActiveRequest = req;
        uSignal DiskWaiting;
    } else {
        PendingClients.insert( &np );                  // insert in ascending order by track number
        uWait np.req;
        ActiveRequest = req;
    } // if
    uWait CurrentRequest;                              // wait for disk to handle request
    return(ActiveStatus);
} // DiskScheduler::DiskRequest

IORequest DiskScheduler::WorkRequest( IOStatus status ) {
    if ( ActiveClient != NULL ) {
        ActiveStatus = status;
        uSignal CurrentRequest;                        // reply to waiting client
        ActiveClient = NULL;
    } // if

    DiskInUse = FALSE;
    if ( !PendingClients.uEmpty() ) {                  // any clients waiting ?
        DiskInUse = TRUE;
        ActiveClient = PendingClients.remove();
        uSignal ActiveClient->req;                     // get work from waiting client
    } else {
        uWait DiskWaiting;                             // wait for client to arrive
    } // if
    // the global variable ActiveRequest is assigned the current request
    return(ActiveRequest);                             // return work for disk
} // DiskScheduler::WorkRequest

void DiskScheduler::Die() {
} // DiskScheduler::Die
```

APPENDIX D μC++ TIMINGS

All timings are run on a Sequent Symmetry S27 (Intel 386, 16Mhz processor), each operation is performed 10,000 times, and both time slicing and runtime checking are turned off. Times are in microseconds.

Uniprocessor

Abstraction \ Operation	create/ delete block	create/ delete dynamic	16 bytes in/ 4 bytes out direct call	16 bytes in/ 4 bytes out accepted call	resume/ suspend cycle	signal/ wait cycle
class	1	33	4	N/A	N/A	N/A
coroutine	60	91	4	N/A	25	N/A
monitor	3	38	16	67	N/A	48
coroutine-monitor	62	96	16	67	25	49
task	111	150	N/A	67	N/A	51

Multiprocessor

Abstraction \ Operation	create/ delete block	create/ delete dynamic	16 bytes in/ 4 bytes out direct call	16 bytes in/ 4 bytes out accepted call	resume/ suspend cycle	signal/ wait cycle
class	1	33	4	N/A	N/A	N/A
coroutine	60	90	4	N/A	25	N/A
monitor	4	39	19	76	N/A	54
coroutine-monitor	63	97	19	75	25	57
task	126	160	N/A	76	N/A	56

REFERENCES

1. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, first edn, 1990.
2. B. Meyer, *Object-oriented Software Construction*, Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.
3. Standardiseringskommissionen i Sverige, *Databehandling – Programspråk – SIMULA*, 1987, Svensk Standard SS 63 61 14.
4. C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpot, 'An introduction to trellis/owl', *SIGPLAN Notices*, 21, (11), 9–16, Nov. 1986.
5. P. Brinch Hansen, 'The programming language concurrent pascal', *IEEE Trans. Softw. Eng.*, 2, 199–206, June 1975.
6. J. G. Mitchell, W. Maybury, and R. Sweet, 'Mesa language manual', Technical Report CSL–79–3, Xerox Palo Alto Research Center, Apr. 1979.
7. R. C. Holt, *Turing Reference Manual*, Holt Software Associates Inc., third edn, 1992.
8. N. Carriero and D. Gelernter, 'Linda in context', *Commun. ACM*, 32, (4), 444–458, Apr. 1989.
9. R. H. Halstead, Jr., 'Multilisp: A language for concurrent symbolic programming', *ACM Trans. Prog. Lang. Syst.*, 7, (4), 501–538, Oct. 1985.
10. W. M. Gentleman, 'Message passing between sequential processes: the reply primitive and the administrator concept', *Software—Practice and Experience*, 11, (5), 435–466, May 1981.
11. R. Hieb and R. K. Dybvig, 'Continuations and concurrency', *SIGPLAN Notices*, 25, (3), 128–136, Mar. 1990, Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming, March. 14–16, 1990, Seattle, Washington, U.S.A.
12. B. N. Bershad, E. D. Lazowska, and H. M. Levy, 'PRESTO: A system for object-oriented parallel programming', *Software—Practice and Experience*, 18, (8), 713–732, Aug. 1988.
13. C. A. R. Hoare, 'Monitors: An operating system structuring concept', *Commun. ACM*, 17, (10), 549–557, Oct. 1974.
14. C. D. Marlin, *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis*, Springer-Verlag, 1980.
15. R. E. Strom, D. F. Bacon, A. P. Goldberg, A. Lowry, D. M. Yellin, and S. A. Yemini, 'Hermes: A language for distributed computing', Technical report, IBM T. J. Watson Research Center, Yorktown Heights, New York, U. S. A., 10598, Oct. 1990.
16. P. A. Buhr and R. A. Stroobosscher, 'The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX', *Software—Practice and Experience*, 20, (9), 929–963, Sept. 1990.
17. United States Department of Defense, *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edn, Feb. 1983, Published by Springer-Verlag.
18. A. N. Habermann and I. R. Nassi, 'Efficient implementation of Ada tasks', Technical Report CMU-CS-80-103, Carnegie-Mellon University, 1980.
19. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, 'An overview of the SR language and implementation', *ACM Trans. Prog. Lang. Syst.*, 10, (1), 51–86, Jan. 1988.
20. N. H. Gehani and W. D. Roome, 'Concurrent C++: Concurrent programming with class(es)', *Software—Practice and Experience*, 18, (12), 1157–1177, Dec. 1988.

21. A. Rizk and F. Halsall, 'Design and implementation of a C-based language for distributed real-time systems', *SIGPLAN Notices*, 22, (6), 83–100, June 1987.

22. H. Bretthauer, T. Christaller, and J. Kopp, 'Multiple vs. single inheritance in object-oriented programming languages. what do we really want?', Technical Report Arbeitspapiere der GMD 415, Gesellschaft Für Mathematik und Datenverarbeitung mbH, Schloβ Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, Deutschland, Nov. 1989.

23. T. A. Cargill, 'Does C++ really need multiple inheritance?', In *USENIX C++ Conference Proceedings*, pages 315–323, San Francisco, California, U.S.A., Apr. 1990. USENIX Association.

24. J. E. Shopiro, 'Extending the C++ task system for real-time control', In *Proceedings and Additional Papers C++ Workshop*, pages 77–94, Santa Fe, New Mexico, U.S.A, Nov. 1987. USENIX Association.

25. P. Labrèche, 'Interactors: A real-time executive with multiparty interactions in C++', *SIGPLAN Notices*, 25, (4), 20–32, Apr. 1990.

26. T. W. Doeppner and A. J. Gebele, 'C++ on a parallel machine', In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, Santa Fe, New Mexico, U.S.A, Nov. 1987. USENIX Association.

27. B. Ciesielski, A. Kreczmar, M. Lao, A. Litwiniuk, T. Przytycka, A. Salwicki, J. Warpechowska, M. Warpechowski, A. Szalas, and D. Szczepanska-Wasersztrum, 'Report on the programming language loglan'88', Technical report, Institute of Informatics, University of Warsaw, Pkin 8th Floor, 00-901 Warsaw, Poland, Dec. 1988.

28. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-oriented Programming in the BETA*, Addison-Wesley, 1993.

29. M. D. Tiemann, 'Solving the rpc problem in gnu C++', In *Proceedings of the USENIX C++ Conference*, pages 343–361, Denver, Colorado, U.S.A, Oct. 1988. USENIX Association.

30. J. D. Ichbiah, J. G. P. Barnes, R. J. Firth, and M. Woodger, *Rationale for the Design of the ADA Programming Language*, Under Secretary of Defense, Research and Engineering, Ada Joint Program Office, OUSDRE(R&AT), The Pentagon, Washington, D. C., 20301, U. S. A., 1986.

31. R. Kamel and N. Gammage, 'Experience with rendezvous', In *Proceedings of the 1988 International Conference on Computer Languages*, pages 143–149, Oct. 1988.

32. C. L. A. Clarke, 'Language and compiler support for synchronous message passing architectures', Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1990.

33. N. H. Gehani and W. D. Roome, *The Concurrent C Programming Language*, Silicon Press, Summit, NJ, 1989.