Adding Concurrency to a Programming Language

Peter A. Buhr and Glen Ditchfield
Dept. of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1
{pabuhr,gjditchf}@plg.uwaterloo.ca

Abstract

A programming language that lacks facilities for concurrent programming can gain those facilities in two ways: the language can be extended with additional constructs, which will reflect a particular model of concurrency, or libraries of types and routines can be written with different libraries implementing different models. This paper examines the two approaches, for object-oriented and non-object-oriented languages. Examples show that concurrency interacts extensively with traditional programming language constructs, and that general elementary facilities for concurrency must be implemented at extremely low levels—the assembly language level, in some cases—and hence that safe support for concurrency requires language extension.

1 Introduction

To take advantage of asynchronous hardware, such as I/O devices or multiple CPUs, a programming language must provide the ability to interact with other programs without blocking, such as calls to operating system I/O routines, or to start multiple independent operations so that if some operations block others can continue to make progress. Traditionally, programming language concurrency has been available only by interacting with the operating system, usually with each task in a different address space. In general, this organizational structure makes creation of new processes and communication among tasks expensive [ABLL92]. Users will not make the additional effort to design and write concurrent programs if the complexity required is too great or the performance pay back is too small. Therefore, new programming languages must provide concurrency and existing programming languages must be augmented with concurrency if they are to be useful in a parallel environment. Finally, concepts that lead up to concurrency, such as coroutines, allow certain kinds of problems, such as finite state machines and push-down automata, to be expressed in eloquent and straightforward ways.

Can concurrency be provided by library definitions built from existing language constructs? If not, what language constructs are needed to make this possible? Would those constructs be useful for purposes other than concurrency? This paper examines programming language facilities that must exist to implement concurrency in object-oriented and in non-object-oriented programming languages. The purpose is to determine if the fundamental aspects of concurrency can be provided through generally available language constructs, or if concurrency requires languages to be augmented with additional constructs.

Much of this analysis comes from our work in adding concurrency to C++, which resulted in a new dialect called μC++ [BS96] that extends C++ with several new language features. μC++ has been criticized for extending C++ instead of adding concurrency using existing language features. This paper attempts to deal with this criticism by showing that it is *not* possible to build concurrency facilities from existing language features in C++ without sacrificing essential features. As well, this discussion should be useful to designers of new languages and those extending existing programming languages with concurrency features. Many ideas presented in this discussion appear in [BDS+92], but this paper presents a more general and thorough analysis.

A general knowledge of concurrency is assumed throughout this discussion.

USENIX C++ Technical Conference, Portland, Oregon, U. S. A., August 1992, pp. 207–224

1.1 Simplicity versus Complexity

We, like others, believe in small languages with strong abstraction facilities.

> In particular, the language should get as much mileage as possible out of its definitional mechanism, never introducing something as a distinct language construct which can better be explained in terms of the definitional mechanism. [Hil83, p. 13]

For example, in C++ [ES90], dynamic memory allocation is provided by library operators new and delete. The default storage management facilities can be replaced by libraries that provide tracing or debugging features [ZH88, Cah], or garbage-collecting allocators [BW88], or allocators tuned to specific allocation patterns. In Pascal [JW85], storage management is provided by standard new and dispose routines. These operations are usually part of the compiler's run-time environment, so replacing them is difficult or impossible. In this respect, C++ is more flexible than Pascal.

Replacing primitives with programmer-definable facilities leads to a smaller, simpler language kernel, and simplicity is generally held to be a virtue in programming languages [Hoa73, Wir74]. However, a small kernel does not imply a reduction in the total complexity of the programming system. Therefore, the main advantage of the library approach is its flexibility. For example, different libraries can provide different models of concurrency, such as the Linda model [CG89] or the Actors model [Agh86]. However, even if a language is general enough to implement a variety of different concurrency models, it is doubtful that programs using different models can interact. Furthermore, issues of syntax and type safety must be dealt with when defining operations like new and delete. At best, a library will be as convenient and as safe as primitive language features. Therefore, as far as a user of a particular model is concerned, there is little difference between an extensible language supporting their model and a specialized language supporting their model.

2 Elementary Execution Properties of Concurrency

As discussed in [BDS+92], there are three elementary execution properties of concurrency:

1. A **thread** sequentially executes programming language statements, independently of and possibly concurrently with other threads. A thread's function is to perform a computation by changing execution-states.

2. An **execution-state** is the state information needed to permit concurrent execution. 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. A programming language determines 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 a continuation. Creating a continuation makes a copy of the current execution-state [HD90].) A **context switch** occurs when a thread switches from one execution-state to another.

3. **Mutual exclusion** is the mechanism that gives a thread sole access to a resource for a period of time.

The first two properties represent the minimum needed to perform execution, and seem to be fundamental in that they are not expressible in machine-independent or language-independent ways. For example, creating a new thread requires creation of system runtime control information, and manipulation of execution-states requires machine specific operations (modifying stack and frame pointers). Mutual exclusion is expressible in terms of simple language statements (for instance by implementing Dekker's algorithm), but doing so is error-prone and computationally expensive, and therefore we believe that mutual exclusion must be provided as an elementary execution property. Therefore, any programming language that supports concurrency must provide primitive constructs to implement these properties. While this can be done in a number of ways, additional design requirements of a programming language may impose additional constraints.

3 Design Options

Concurrency facilities must blend with other aspects of a programming language.

form of a task – Does a task resemble other language constructs? A non-object-oriented programming language might present a task as an independently executing program, analogous to the body of a non-concurrent program. An object-oriented programming language might present a task as an object, with an interface defined by a set of member functions.

form of communication – Does task communication resemble other language communication? Task communication might resemble a routine call, with data passed as arguments and received as parameters, or tasks might communicate through intermediate "channel" objects, which resembles file I/O. Alternately, communication could involve new operations to send and receive "message" objects, which programs must assemble and disassemble. Multiple forms of communication in a language can be confusing for users and inefficient because data must be transformed from one form to another along a communication path. We argue for using the routine-call mechanism in C++ because that is the form used to communicate with objects.

static type-checking – Can all communication in the language be statically type-checked? Languages with compile time (static) type-checking versus runtime (dynamic) type-checking have additional requirements on the communication mechanism. Sufficient definitions must be made and be available so that the compiler can type check all communication, especially across separate translation units.

declaration scopes – Are the concurrency features available or restricted by the declaration scopes of the language? For instances, if tasks resemble objects, then it should be possible to declare a task anywhere that an object can be declared.

A designer of concurrency facilities must choose between alternative ways of providing them.

direct and indirect communication –

Can tasks communicate directly with one another or does all communication occur through a third party? If communication requires a third party, e.g., a monitor [MMS79, Hol92] or tuple space [CG89], this can slow execution when a large number of tasks are interacting in a complex way because of additional synchronization and data transfers with the intermediate object.

synchronization and mutual exclusion – Is mutual exclusion and synchronization implicit and limited in textual scope, or explicit and tied to the flow of control? It is our experience that requiring users to build complex mutual exclusion facilities, like monitors, out of low-level mutual exclusion primitives, like locks, often leads to incorrect programs. Furthermore, we have noted that reducing the textual scope in which synchronization occurs reduces errors in concurrent programs.

synchronous or asynchronous communication – Both synchronous and asynchronous communication are needed in a concurrent system. In synchronous communication, a task that transmits information suspends execution until another task receives it and replies; in asynchronous communication, the transmitter may continue execution before the receiver picks up the data. Since synchronous communication can implement asynchronous and vice versa, a language need only provide one of the two mechanisms. We argue that a language should provide synchronous communication out of which asynchronous communication can be built. If asynchronous communication is the primitive mechanism, this usually implies the existence of variable-sized dynamically-allocated buffers. In general, this is too expensive a mechanism to be built into the language. Asynchronous mechanisms should be provided by library facilities like buffers and/or futures.

order of processing requests – An object that is accessed concurrently must have control over the order in which it services requests. Without this ability, all requests must be processed in first-in first-out (FIFO) order; any other order requires a programmer to devise a multi-step protocol. FIFO servicing may inhibit concurrency and has deadlock problems [Gen81], while protocols are error-prone because a user may not obey the protocol (e.g., never retrieve a result). The ability to postpone a request is sufficient, where postponing means that a task can accept a request, examine it, and decide not to perform it for an unspecified time, while continuing to accept new requests (available in Thoth [Che82], Harmony [Gen85], and the V-system [Che88]).

It is also extremely convenient and often more efficient if a concurrent object can also control which pending request it receives next (available in SR [AOC$^+$88] and Concurrent C [GR89]), rather than having to receive requests in FIFO order and possibly postpone inappropriate ones. There are many situations where a concurrent object knows that it can service only a certain kind of request or a request from a certain object next. However, the ability to select a pending request is insufficient if servicing requires other resources that may not be immediately available; only the ability to postpone a request allows a task to continue servicing requests until the resources becomes available and the request can be completed.

We reject any solutions to these options that involve coding conventions or multi-step protocols because such solutions are error-prone both to implement and maintain.

4 Concurrency Libraries

The following sections examine the difficulties in adding concurrency through language definitions and routines assuming that some primitive mechanisms *already* exists to provide the three elementary execution properties. Both object-oriented and non-object-oriented programming languages are examined.

4.1 Starting a Library

Currently, C++ does not define the relative order of initialization of objects declared with static storage duration in different translation units. As a result, there is no way to ensure that a library is initialized before the objects that depend on it are instantiated (e.g., like the runtime system of a concurrency library). This leaves a library implementor with three options in C++:

1. Forbid the declaration of library objects with static storage duration.
2. Test repeatedly in the library to ensure that initialization is performed.
3. Declare an instance of a library initialization object before any declarations that depend on the library in each translation unit, but ensure that only one of the initialization instances actually performs the initialization of the library. The declaration of the initialization object can be made implicit by putting it in the include file for the library, which must be included in each translation unit before library features are used. (This is the approach used to start the μC++ runtime library.)

The first solution is very restrictive, the second is inefficient, and the third is a C++ idiom that is not obvious to a library implementor. This situation is handled properly by modules in other languages [Uni83, CDG$^+$88], which define an order of initialization among modules.

4.2 Context Switching

C and C++ have a language facility called setjmp/longjmp that was introduced to provide a simple form of exceptional control flow. These routines save and restore execution-state to allow non-local gotos. This language feature has been used as the basis for context switching in some thread packages.

However, setjmp/longjmp can be inefficient for context switching. The problem occurs with co-processor data, such as floating point registers, and any other data that is task specific. Saving all of this data substantially increases the cost of context switches. For example, our test results show that on a Sequent Symmetry S27 (Intel 386) the context switch time can double when the floating-point registers are saved. Since many tasks do not use floating point, this can be of significant concern. Flow analysis in a compiler may determine that a task does not use the floating point registers so only the fixed point registers have to be saved on a context switch. Most light-weight tasking systems require the user to explicitly indicate whether the floating point registers should be saved on a context switch, which is error prone.

4.3 General Library Routines

In general, most UNIX library routines are *not* reentrant. For example, many random number generators maintain an internal state between successive calls, and there is no mutual exclusion on this internal state. Therefore, one task that is executing the random number generator can be pre-empted and the generator state can be modified by another task. This can result in problems with the generated random values or errors. One solution is to supply cover routines for each non-reentrant routine that guarantee mutual exclusion on calls, but this is not practical as too many cover routines have to be created.

Part of this problem can be handled by allowing pre-emption only in user code. When a pre-emption occurs, the handler for it checks if the current task is executing user code. If it is, the handler causes a context switch to another task. If the current task is not executing user code, the interrupt handler resets the timer and returns without rescheduling another task. In theory, a task that calls system routines at fortunate moments might never be pre-empted.

Determining whether an address is in user code is done in μC++ by relying on the linker to place programs in memory in a particular order. μC++ programs are compiled using a command that invokes the C++ compiler and includes all necessary include files and libraries. The command forces the linker to bracket all user modules between two precompiled routines, uBeginUserCode and uEndUserCode. The pre-emption interrupt handler simply checks if the interrupt address is between the addresses of uBeginUserCode and uEndUserCode to determine if the interrupt occurred in user code. This approach assumes that all libraries are non-pre-emptable, which inhibits concurrency for those routines that are reentrant (e.g., sin, cos, etc.).

Allowing pre-emption only in user code is sufficient to deal with non-reentrant routines on uniprocessors. On multiprocessors, we rely on the vendor to provide reentrant routines (which is not always a reasonable assumption). In the future, all library routines will have to be reentrant.

4.3.1 I/O Libraries

The standard I/O libraries provide an example of undesirable interactions between libraries. To ensure maximum parallelism in light-weight tasking systems, it is desirable that a task not execute an operation that causes the processor it is executing on to block. UNIX I/O operations can be made to be nonblocking, but this requires special efforts since the I/O operations do not restart automatically when the operation completes. Instead, it is necessary to poll for I/O completions, and possibly block the program if all tasks are directly or indirectly blocked waiting for I/O operations to complete. Since this is complex, most concurrency libraries provide nonblocking versions of the I/O routines.

In μC++, I/O cover objects exist for the I/O streams, which check the ready queue before performing their corresponding C++ I/O operations. If no tasks are waiting to execute, blocking can occur because all tasks in the system must be directly or indirectly waiting for an I/O operation to complete. If tasks are waiting, a nonblocking I/O operation is performed. One of the tasks performing an I/O operation polls for completion of any I/O operation and yields control of the processor if no I/O operation has completed. When an I/O operation completes, all the I/O tasks are unblocked and each checks if its I/O operation has completed. This scheme allows other non-I/O tasks to make progress with only a slight degradation in performance due to the polling task.

4.4 Abnormal Event Handling

Handling abnormal events cannot be done properly using library mechanisms, such as return-codes, because these mechanisms do not scale to large robust systems. Virtually all new programming languages provide language facilities like exceptions to deal with abnormal situations. Concurrency adds another dimension to the handling of abnormal events in a program. How does exception handling work when there are multiple execution-states? How are hardware and software interrupt facilities introduced? In [BMZ92], abnormal event handling mechanisms for concurrent environments were extensively analyzed. Two distinct mechanisms were identified:

1. An *exceptional* change in control flow, using the stack of an execution-state (i.e., C++ style exceptions).

2. A corrective action by an *intervention* in the normal computation of an operation, which includes interrupt/signal handling between tasks.

Facilities to support synchronous exceptions, and synchronous and asynchronous interventions were implemented though a library facilities in C. Unfortunately, the resulting library facilities burden users with both syntactic and semantic details, coding conventions and protocols. Our conclusion after constructing an abnormal event library is that it is impossible to provide powerful abnormal event handling mechanisms without augmenting the programming language. C++ was extended with exceptions for the same reason [KS90].

4.5 Task Libraries for Object-Oriented Languages

In an object-oriented language, the natural way to provide concurrency through a library is to define an abstract class, Task, that implements the task abstraction. The constructor for Task creates a thread to "animate" the task. User-defined task classes inherit from Task, and tasks are objects of these classes. This approach has been used to define C++ libraries that provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88].

When this approach is used, task classes should have the same properties as other classes, so inheritance from task types should be allowed. Similarly, tasks should have the same properties as other objects. This latter requirement suggests that tasks should communicate via calls to member routines, since ordinary objects receive requests that way, and since the semantics of routine call matches the semantics of synchronous communication nicely. The body of the task (that is, the code that is executed by the thread associated with a task) has the job of choosing which member routine call should be executed next.

The following are the stages that a library package must deal with during the lifetime of a task:

1. thread creation for the task
2. task initialization
3. task body execution, which controls most of the synchronization with other tasks
4. task termination
5. joining/synchronization by another task with a task that is terminating

These stages will be referred to in the following sections.

4.5.1 Task Body Placement

The body of a task must have access to the members of a task, and the Task constructor must be able to find the body in order to start the task's thread running in it. Therefore, in the library approach, the task body must be a member of the task type. At first glance, the task's constructor seems like a reasonable choice. However, the requirement that it be possible to inherit from a task type forbids that choice. Let T1 be a task type, with a constructor that contains initialization code for private data and the task body. Now consider a second type T2 that inherits from T1. T2's constructor must specify a new task body: it must somehow override the task body in T1's constructor, but still execute the private data initialization code. Given that they are contained in the same block of code, that is clearly impossible.

The correct solution is to put the body in a special member routine, perhaps called main. main would be declared by Task, and would be a virtual routine so that task types could replace it.

4.5.2 Thread Creation

When one task creates another, the creating task's thread executes statements in Task's constructor that create a new thread. The library implementor must decide which thread does what jobs. The approach that produces the greatest concurrency has the new thread execute the new task's constructors and body, while the creating thread returns immediately to the point of the declaration of the object. However, the normal implementation of constructors in C++ makes this difficult or impossible if inheritance from task types is allowed. Each constructor starts by calling the constructors of its parent classes. By the time Task's constructor is called, there can be an arbitrary number of routine activations on the stack, one for each level of inheritance. It is not possible for the initialization code for Task to examine the stack to locate the return point for the original constructor. Only compiler support, such as marking the stack at the point of declaration or passing implicitly the return address for the creating thread up the inheritance chain, can make this approach work. In the absence of compiler support, the creating thread must execute the new task's constructors, while the new thread executes the task body, which inhibits concurrency somewhat.

4.5.3 Task Initialization and Execution

The next problem results from an interaction between task initialization and task body execution. The task's thread must not begin to execute the task body until after the task's constructor has finished. However, in the library approach, the code to start the thread running in the task body appears in Task's constructor. In C++, that code is executed first, *before* the constructors of any derived classes. Hence the new thread must be created in the "blocked" state, and must be unblocked after the derived constructors finish. A second, more subtle problem, results from the semantics of initialization. While Task's constructor is executing, the new task is considered to be an instance of class Task, not the actual task class being instantiated. This means that, within Task's constructor, the virtual main routine that contains the task's body is inaccessible; calling main in Task's constructor will not execute the correct task body!

PRESTO dealt with this problem by requiring an explicit action to unblock the thread. In this approach, the Task class provides a start() member routine that must be called after the declaration of a task, but before any calls to the task's member routines. At that point the constructors have all finished and main refers to the actual task body. This two-step creation protocol opens a window for errors: programmers may fail to start their tasks.

A similar interaction exists between task body execution and task termination. When one task deletes another, it will call the deleted task's destructor. The destructor must not begin execution until after the task body has finished. However, the code that waits for the task body to finish cannot be placed in Task's destructor, because it would be executed last, *after* the destructors of any derived classes. Task designers cannot simply move the task's termination code from the destructors to the end of the task body, because that would prevent further inheritance: derived classes would have no way to execute their base class's termination code. Task could provide a finish() routine, analogous to start(), which must be called before task deletion, but this two-step termination protocol is even more error-prone than the creation protocol.

A general language mechanism like Simula's inner [Sta87] would solve these problems. In a single inheritance hierarchy, an inner statement in a constructor (or destructor) of a base class acts like a call to the constructor (or destructor) of the derived class. For instance, given

```
class T1 {
  public:
    T1() { s1; inner; s2; };
};
class T2: public T1 {
  public:
    T2():T1() { s3; };
};
T1 a_t1;
T2 a_t2;
```

the initialization of a_t1 executes statements s1 and s2, and the initialization of a_t2 executes s1, s3, and s2, in that order. (T2::T2 might also contain inner statements, which would invoke the constructors of classes derived from T2.) In T1::T1, before the inner statement, a_t2 is considered to be an instance of class T1. After the inner statement, it is an instance of T2, and the meaning of calls to virtual routines changes accordingly.

A concurrency library would use inner in Task to control the timing of events. Task's constructor would use an inner statement to execute the derived task class's constructors and establish the proper meaning for main, and then create the new thread running (unblocked) in main. Task's destructor would wait for the task body to finish, and then would use inner to execute the task class's destructors. However, with this technique the creating thread executes the constructors, which inhibits concurrency as mentioned in section 4.5.2.

The inner statement is useful, independent of concurrency, for fine control during the initialization of an object. For instance, it lets constructors of base classes call virtual functions that are redefined by derived classes. However, it is not obvious how inner could be added to C++. inner must invoke constructors and destructors in the order defined by C++, taking multiple inheritance and virtual inheritance into account.

Furthermore, a class can have many constructors, and descendants specify which of their base class's constructors are called. Finally, there should be some canonical translation from inner to efficient, standard C.

4.5.4 Task Communication

Communication among tasks also presents difficulties. In library-based schemes (and some languages), it is done via message queues, called ports or channels [And91]. 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. Inheritance from a Message class could be used, instead of a union, but the task would then have to perform type tests on messages before accessing them with facilities like Simula's is and qua. Currently, runtime type-tests are counter to the C++ design philosophy.

If multiple queues are used, a library facility analogous to the Ada [Uni83] select statement is needed to allow a task to wait for messages to arrive on more than one queue. However, building a library facility similar to a select statement requires λ-expressions (anonymous nested routine bodies) or preprocessor macros to support the blocks of code that may or may not be invoked depending on the selection criteria, for example:

```
select( accept(queue-name,code-body ) || accept( ..., ... ) || ... );
```

where the *code-body* is a λ-expression and represents the code executed after a particular message queue is accepted. This capability is essential so that a particular action can be performed after a message is received. Furthermore, there is no statically enforceable way to ensure that only one task is entitled to receive messages from any particular queue, for example:

```
MsgQueueType A;
MsgQueueType B;

class TaskType : public Task {
    void main() {          // task body
        ...
        select( accept( A, NULL ) || accept( B, NULL ) );
        ...
    }
};

TaskType T1, T2;
```

Tasks T1 and T2 simultaneously accept messages from the same queues. While it is straightforward to check for the existence of data in the queues, if there is no data, both T1 and T2 must wait for data to appear on either queue. To implement this, tasks have to be associated with both queues until data arrives, given data when it arrives, and then removed from both queues. This implementation would be expensive since the addition or removal of a message from a queue would have to be an atomic operation across all queues involved in a waiting task's accept statement to ensure that only one data item from the accepted set of queues is given to the accepting task. In languages with concurrency support, the compiler can disallow accepting from overlapping sets of message queues by restricting the select statement to queues the task declares. Compilers for more permissive languages, like SR [AOC+88], perform global analysis to determine if tasks are receiving from overlapping sets of message queues; in the cases where there is no overlap, less expensive code can be generated. In a library approach, access to the message queues must assume the worst case scenario.

If the routine-call mechanism is to be used for communication among tasks (as in μC++), a select statement again requires a λ-expressions or preprocessor macros. Furthermore, each public member routine has to have special code at the start and possibly at the exits, which the programmer has to provide by following a convention. This special code would provide, at the least, mutual exclusion and control selective entry. Object-oriented programming languages that support inheritance of routines, such as LOGLAN'88 [CKL+88] and Beta [MMPN93], can provide special member code automatically. (The use of inner in a constructor is a special case of routine inheritance, where the derived class's constructor inherits from the base class's constructor.) Whatever the mechanism, it must allow the special code to be selectively applied to the member routines. For example, there are cases where not all public member routines require mutual exclusion and where some private members require mutual exclusion. In languages with concurrency support, the compiler can easily disallow accepting another task's member, so the problem of accepting from overlapping sets of members will not occur.

4.6 More Inheritance Problems

Regardless of whether a concurrency library or language extensions are used to provide concurrency in an object-oriented language, new kinds of types are introduced, like coroutine, monitor, and task. These new kinds of types complicate inheritance. The trivial case of single inheritance among homogeneous kinds, i.e., a monitor inheriting from another monitor, is straightforward because any implicit actions are the same throughout the hierarchy. (An additional requirement exists for tasks: there must be at least one task body 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 task body or in redefined virtual routines.

Inheritance among heterogeneous types can be both useful and confusing. Having classes with mutual exclusion inherit from classes without it is useful to generate concurrent types from existing non-concurrent types. For instance, a sharable queue task could be defined by inheriting from an ordinary queue and redefining all of the class's member routines to provide mutual exclusion.

```
class Queue {
  public:
     void insert( ... ) ...
     virtual void remove( ... ) ...
};
class MutexQueue : public Queue, public Task {
     virtual void insert( ... ) ...
     virtual void remove( ... ) ...
};
```

However, this example demonstrates the dangers caused by non-virtual routines.

```
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
```

Queue::insert does not provide mutual exclusion because it is a member of Queue, while 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; 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 C++'s private inheritance because no subtype relationship is created and hence the assignment to qp would be invalid.

Heterogeneous inheritance among entities like monitors, coroutines and tasks can be very confusing. While it is always possible to construct some meaning for such inheritance, we reject it for the following reason. Classes are written as ordinary classes, coroutines, monitors, or tasks, and we do not believe that the coding styles used in each can be arbitrarily mixed. For example, an instance of a task class that inherits from an ordinary class can be passed to a routine expecting instances of the class. If the routine calls one of the object's member routines, it could inadvertently block 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.

Multiple inheritance simply exacerbates the problem stated above and it significantly complicates the implementation, which slows the execution. For example, accepting member routines is significantly more complex with multiple inheritance because it is not possible to build a static mask to test on routine entry. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [BCK89, Car90].

4.7 Libraries for Non-Object-Oriented Languages

What are the problems of adding concurrency to non-object-oriented languages and can it be done using a library approach?

We have extensive experience in adding concurrency to C using the library approach in a system called the μSystem [BS90]. The μSystem is a light-weight tasking library for C that runs on the following processors: M68K, NS32K, VAX, MIPS, i386/486, Sparc, and the following UNIX operating systems: Apollo SR10 BSD, Sun OS 4.x, Tahoe BSD 4.3, Ultrix 3.x/4.x, DYNIX, Umax 4.3, IRIX 3.3. As well, the μSystem provides coroutines, several forms of synchronization and communication, and synchronous and asynchronous abnormal event handling features. We can say with authority that many of the design criteria stated earlier cannot be provided without language extensions. This statement is true for all light-weight tasking systems for C using a library approach [Che82, Gen85, Che88, Sun88, Enc88, CG89]. Only because C allows extensive violations of its type system is it possible to build an adequate set of library facilities. In general, either the type system is violated or the functionality is restricted, as illustrated in the following examples. In the library approach, a task is formed by starting a thread executing in a routine body. If this routine is allowed to have arbitrary parameters, there is no type safe way to pass arguments in a library approach. If the routine is not allowed to have parameters or has a fixed parameter list, this complicates initialization as a protocol is now required between the creator and the new task to pass the initialization values that cannot be provided when the task is started. Type-safe direct communication among tasks is impossible in a library approach because a routine has only one entry point that can be invoked in a type safe way. Type-safe indirect communication is possible through a monitor library, but a library monitor requires user conventions for proper usage. These problems exist in thread packages in other languages, such as Modula-2.

Non-object-oriented languages with language support for concurrency [MMS79, Hol92] can pass initial arguments to a new task in a type-safe way because a special statement to start the new task is provided. However, these languages cannot solve the direct communication problem, and hence, do not provide direct communication. Communication is normally indirect through monitors, which may be part of the language so that user conventions are unnecessary.

In [Hil83, pp. 136–144], Hilfinger outlines how concurrency might have been added to the programming language Ada using elementary programming language constructs instead of high-level programming language constructs (Ada's existing concurrency facilities are largely class-based). In Hilfinger's proposal, creating threads and execution-states are still elementary properties [Hil83, pp. 141–142] that are tied into the language's runtime environment at a very low-level. λ-expressions and routine variables (pointers to routines), which Ada does not have, are required to build a select statement. Furthermore, without automatic dereferencing of pointer variables, usage syntax would be unacceptable. Finally, users are required to follow coding conventions for each routine that requires mutual exclusion. Inheritance problems do not exist because Ada's type system does not have inheritance. (The Ada 9X proposals has a large number of new language features to support concurrency rather than building on existing language features.) With all its faults, Hilfinger's outline does support type-safe direct communication. Appendix A presents our minimalistic approach for including concurrency into a non-object-oriented language that supports type-safe direct communication. Our solution requires language support to achieve all the design options stated earlier.

5 Conclusion

Our main conclusion is that concurrency is a fundamental aspect of a programming language that cannot be built easily from primitive non-concurrent language constructs. Creation of a thread and execution-state cannot be implemented from basic constructs without working at a very low-level, possibly violating both type-safety and the integrity of the runtime environment, nor can mutual exclusion be implemented inexpensively. Even given the three elementary properties, library facilities do not usually integrate into the language's type system, often requiring protocols and coding conventions, and library facilities may be inefficient. The purpose of high-level languages is to automate protocols and conventions, and provide global optimization to make a program efficient.

The following are some specific observations:

- If a language supports all the design options presented at the beginning, a large number of different models of concurrency can be implemented. Usually, the expressive power of the language is the limiting factor as to how well a model can be expressed.
- Minimizing the cost of a context switch requires language support because only the compiler knows exactly how much state a particular object is using.
- Without language support, object-oriented languages with inheritance have problems in coordinating initialization and the starting of a task's new thread, as well as specifying the location where the thread starts execution.
- Indirect type-safe communication using message queues, where selection can occur from multiple overlapping message queues, is costly to implement.
- Direct type-safe communication requires an aggregating construct with multiple entry points. Therefore, class-based programming languages appear to be able to support direct type-safe communication better than non-object-oriented languages. This advantage results from the special relationship between the class and its member routines, which can be extended with other properties like mutual exclusion.
- If an object-oriented language provides special type-specifiers, like task and monitor, there are addition problems with heterogeneous inheritance among the type specifiers, e.g., a class which inherits from a monitor which inherits from a task.
- A language's exception handling mechanism must be designed to work with its concurrency mechanism because exceptions work with the stack associated with an execution-state (exceptions search the execution stack) and there are now multiple execution-states. Furthermore, a mechanism to deal with interrupts must be provided.

A Concurrency in Non-Object-Oriented Programming Languages

In non-object-oriented languages, data structures corresponding to objects are created by instantiation of pure types (i.e., types containing only data fields). These types are grouped together with the routines that manipulate the data structures into collections such as modules; we refer to these languages as routine/type/module (RTM) languages (e.g., Ada, Modula-2 [Wir85]). Since a module does not define a type, subtyping is not a concept that pertains to modules. Instead, polymorphism/reuse is accomplished through overloading, parameter generalization, and call-site inferencing and binding [CW90]. Data structures in RTM languages can be manipulated like objects using the following conventions. A module exports an opaque type, which corresponds to the class type. The opaque type provides encapsulation and information hiding outside the module but the module routines can access all the fields of the type for instances passed as arguments. Initialization and termination code can be associated with an opaque type by overloading the routines new and delete, which are called implicitly after allocation and before deallocation of a data item. (We do not know of any RTM language that supports this facility, but it is possible if the language's polymorphism capabilities permit overloading.) Therefore, any class can be transformed into an opaque type in a module.

Before discussing how concurrency can be added to RTM languages, we want to dismiss approaches that use modules to mimic certain kinds of objects, e.g., monitor modules [MMS79, Hol92], as they are not general. In this approach, a module creates a single object and for a task, execution would begin in the module initialization code. However, since a module is instantiated only once, it does not allow creation of multiple instances of the object. Generic modules can mitigate this problem, but we feel that using generics to generate multiple instances of the same type is an inappropriate use of this facility.

When opaque types are used to define classes, calls to module routines are normally unsynchronized, that is, the caller's state is saved and control transfers to the called routine. However, this is not the case for routines that operate on monitor or task types (or persistent types [BZ88]). Special call-action, namely that required to implement mutual exclusion and/or task synchronization, must be provided by routines that directly access the fields of these types. Furthermore, creation of instances of the module's opaque type may also have to start a new thread of control at a particular location. When a language provides special constructs, e.g., task construct, the special call-action is implied; however, it must be explicitly stated when using RTM languages.

Figure 1 shows a possible implementation of a bounded buffer as a class-based task and a RTM task. The implementation of the RTM task preserves all the initial design options so the comparison is fair. First, a task record's semantics are that instantiation creates a new thread and the thread begins execution in the overloaded routine new that has a parameter of the corresponding record type. The opaque facility provides information hiding capability. (The overloaded routine delete would be called implicitly to perform termination after the task's thread has terminated and before deallocation.) Second, a routine with a task record parameter implies that a task making a call to it is blocked if the corresponding argument is currently being accessed by another task. In that case, the caller must be put on one or one of a number of hidden queues associated with the argument corresponding to that task record parameter; this would result in an efficient implementation. When the routine finishes, the task argument is released and is available for access by other tasks. A routine can only have one parameter that requires mutual exclusion, since putting a task on multiple queues does not make sense. Because of this restriction, having multiple task records defined in a single module would be done solely for organizational reasons as no routine could access both task records. If an exported module routine simply wants to pass the task record parameter on to another routine without requiring mutual exclusion, it must qualify the parameter type with a nomutex qualifier. Lastly, our example allows a task to be blocked using wait and signal statements and condition variables as for a task created using a class-based task.

Any routine written outside of the module that has a parameter of a module's task-record type would not acquire mutual exclusion (the parameter would be implicitly nomutex). The following problems arise if this rule is not adopted. First, the implementation of a task record could not use multiple entry queues because a new queue would have to be added to the task-record type for the new routine and this would require extending the record, which cannot be done. At best, a single queue must be used and it must be searched when an accept is executed; this might substantially affect performance. Second, the new routine cannot access any of the opaque fields of the task data structure, except indirectly through the routines provided in the module, hence there is little point in obtaining mutual exclusion. As a result, parametric polymorphism techniques cannot be used to provide code reuse.

If an RTM language supported record concatenation [Wir88] that is applicable to task-record types, this being analogous to object-oriented inheritance, this would result in equivalent implementation with a class-based task, and hence, the same performance. In this case, a new module would define a task-record type that is an extension of an existing task-record type from another module. For each routine in the new module with a task record parameter of the extended task-record type, a new entry queue would be added to the extended type.

We believe that this outline for RTM tasks is probably the best way to extend RTM languages to support types with special semantics. The drawback is that special semantics actions must be explicitly specified, e.g. nomutex clause, by the user and its implementation might imply some restrictions on the polymorphism mechanism or vice versa.

References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[And91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

Class-based Task	RTM-based Task

<pre>task buffer {
    const int QSize = 3;
 	
    int front, back, count;
    int queue[QSize];
 	
  protected:
    void main() {
        front = back = count = 0;
 	
        for (;;) {
            accept(stop)
                break;
            or when (count != Qsize)
                accept(insert);
            or when (count != 0)
                accept(remove);
        }; // for
    }; // main
public:
    void stop() {
    }; // stop
 	
    void insert(int elem) {
        queue[back] = elem;
        back = (back + 1) % QSize;
        count += 1;
    }; // insert
 	
    int remove() {
        int elem;
 	
        elem = queue[front];
        front = (front + 1) % QSize;
        count -= 1;
        return(elem);
    }; // remove
}; // buffer
 	
buffer a, b, c;
 	
a.insert(1);
i = a.remove();</pre>
	<pre>module buffermodule {
	    export buffer, new, stop, insert, remove;
	    const int QSize = 3;
	    opaque type buffer = task record {
	        int front, back, count;
	        int queue[QSize];
	    }; // buffer
	
	    void new(buffer b) {
	        b.front = b.back = b.count = 0;
	
	        for (;;) {
	            accept(stop)
	                break;
	            or when (count != Qsize)
	                accept(insert);
	            or when (count != 0)
	                accept(remove);
	        }; // for
	    }; // new
	
	    void stop(buffer b) {
	    }; // stop
	
	    void insert(buffer b, int elem) {
	        b.queue[back] = elem;
	        b.back = (b.back + 1) % QSize;
	        count += 1;
	    }; // insert
	
	    int remove(buffer b) {
	        int elem;
	
	        elem = b.queue[front];
	        front = (front + 1) % QSize;
	        count -= 1;
	        return(elem);
	    }; // remove
	}; // buffermodule
	
	buffer a, b, c;
	
	insert(a, 1);
	i = remove(a);</pre>

Figure 1: Bounded Buffer using Class-based Task and RTM-based Task

[BCK89] Harry Bretthauer, Thomas Christaller, and Jürgen 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, November 1989.

[BDS+92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BLL88] 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, August 1988.

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software—Practice and Experience*, 20(9):929–963, September 1990.

[BS96] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.6. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1996. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.

[BW88] Hans-J. Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[BZ88] P. A. Buhr and C. R. Zarnke. Nesting in an Object Oriented Language is NOT for the Birds. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object Oriented Programming*, volume 322, pages 128–145, Oslo, Norway, August 1988. Springer-Verlag. Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.

[Cah] Conor P. Cahill. debug_malloc. comp.sources.unix, volume 22, issue 112.

[Car90] T. A. Cargill. Does C++ Really Need Multiple Inheritance? In *USENIX C++ Conference Proceedings*, pages 315–323, San Francisco, California, U.S.A., April 1990. USENIX Association.

[CDG+88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report. Technical Report 31, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, August 1988.

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Che82] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier, 1982.

[Che88] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CKL+88] Boleslaw Ciesielski, Antoni Kreczmar, Marek Lao, Andrzej Litwiniuk, Teresa Przytycka, Andrzej Salwicki, Jolanta Warpechowska, Marek Warpechowski, Andrzej Szalas, and Danuta Szczepanska-Wasersztrum. Report on the Programming Language LOGLAN'88. Technical report, Institute of Informatics, University of Warsaw, Pkin 8th Floor, 00-901 Warsaw, Poland, December 1988.

[CW90] G. V. Cormack and A. K. Wright. Type-dependent Parameter Inference. *SIGPLAN Notices*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

[DG87] Thomas W. Doeppner and Alan J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association.

[Enc88] Encore Computer Corporation. *Encore Parallel Thread Manual, 724-06210*, May 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, first edition, 1990.

[Gen81] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software—Practice and Experience*, 11(5):435–466, May 1981.

[Gen85] W. Morven Gentleman. Using the Harmony Operating System. Technical Report 24685, National Research Council of Canada, Ottawa, Canada, May 1985.

[GR89] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. *SIGPLAN Notices*, 25(3):128–136, March 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming, March. 14–16, 1990, Seattle, Washington, U.S.A.

[Hil83] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design*. ACM Distinguished Dissertations. MIT Press, 1983.

[Hoa73] C. A. R. Hoare. Hints on Programming Language Design. Technical Report CS-73-403, Stanford University Computer Science Department, December 1973. Reprinted in [Was80].

[Hol92] R. C. Holt. *Turing Reference Manual*. Holt Software Associates Inc., third edition, 1992.

[JW85] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer–Verlag, third edition, 1985. Revised by Andrew B. Mickel and James F. Miner, ISO Pascal Standard.

[KS90] Andrew Koenig and Bjarne Stroustrup. Exception Handling in C++. *Journal of Object-Oriented Programming*, 3(2):16–33, July/August 1990.

[Lab90] Pierre Labrèche. Interactors: A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Notices*, 25(4):20–32, April 1990.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA*. Addison-Wesley, 1993.

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979.

[Sho87] Jonathan 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, November 1987. USENIX Association.

[Sta87] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14.

[Sun88] *System Services Overview, Lightweight Processes*, chapter 6, pages 71–111. Sun Microsystems, May 1988. available as Part Number: 800-1753-10.

[Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.

[Was80] Anthony I. Wasserman, editor. *Tutorial: Programming Language Design*. Computer Society Press, 1980.

[Wir74] Niklaus Wirth. On the Design of Programming Languages. In *Information Processing 74*, pages 386–393. North Holland Publishing Company, 1974. Reprinted in [Was80].

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, third, corrected edition, 1985.

[Wir88] N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[ZH88] Benjamin Zorn and Paul Hilfinger. A Memory Allocation Profiler for C and Lisp Programs. In *Summer 1988 USENIX proceedings*, 1988.