

DIRECT INQUIRIES TO:
INDIANA UNIVERSITY
ACCOUNTS PAYABLE
812-855-4004



PAYEE OR VENDOR NAME			PAYEE IDENT. NUMBER	CHECK DATE	CHECK NUMBER
UNIVERSITY OF WATERLOO			000010192 00	06/05/89	155003
VENDOR'S INVOICE NO.	VENDOR'S CUST. NO. FOR I.U.	INVOICE DATE	PURCHASE ORDER NO.	ACCOUNTS PAYABLE	INVOICE AMOUNT
NA015065890602 TERMS -	PREPAID	890531	109050156	AP239248	2.00
<i>rec'd cheque & sent report June 12/89</i>			NET AMOUNT	OF CHECK	*****2.00



INDIANA UNIVERSITY PURCHASE ORDER

PAGE 01

P.O. NUMBER: 10905-0156
Env # NA 015065870602

VENDOR: UNIVERSITY OF WATERLOO DEPT. OF COMPUTER SCIENCE ATTN: RESEARCH REPORT SECRETARY WATERLOO, N2L 3G1 ONTARIO,		SHIP TO ADDRESS: INDIANA UNIVERSITY/10905-0156 CENTRAL RECEIVING DEPARMENT 11TH & WALNUT GROVE BLOOMINGTON, IN 47405
ORDER DATE: 05-31-89	I.U. CUSTOMER NO: 	BILL TO ADDRESS: INDIANA UNIVERSITY ACCOUNTS PAYABLE POST OFFICE BOX 4040 BLOOMINGTON, IN 47402
TERMS:	FOB:	

[illegible]

GUY J. DE STEFANO

UNIVERSITY DIRECTOR OF PURCHASING

PURCHASING AGENT: RAY E NEW

812-855-8752

1. ACCEPTANCE OF THIS ORDER IS REQUESTED STATING DELIVERY DATE.
2. UNLESS OTHERWISE INSTRUCTED, ALL LOCAL TRUCK DELIVERIES ARE TO BE MADE AT THE RECEIVING DEPT. CLINT STORES BLDG. 11TH ST AND WALNUT GROVE
B. 12 OR 1-5 MONDAY THRU FRIDAY.
3. SEND ALL CORRESPONDENCE MARKED WITH ORDER NUMBER, ATTENTION: PURCHASING, TO THE PURCHASING DEPARTMENT, P.O. BOX 3040.
4. INDIANA UNIVERSITY IS FREE OF ALL EXCISE AND INDIANA SALES TAXES. CERTIFICATE #35-0001673-0000 WILL BE FURNISHED UPON REQUEST.
5. PLACE ORDER NUMBER ON ALL BILLS AND PACKAGES. DO NOT CONSOLIDATE SHIPMENT OF ORDERS UNLESS EACH ORDER IS SEPARATED BY ORDER NUMBER AND
SHIPPING CONTAINER SHOWS ALL ORDER NUMBERS INCLUDED.
6. INDIANA UNIVERSITY POLICY PROHIBITS DISCRIMINATORY PRACTICES IN ALL PHASES OF EMPLOYMENT WITHOUT REGARD TO RACE, COLOR, RELIGION, SEX OR
NATIONAL ORIGIN. THE CONTRACTOR OR VENDOR AGREES WITH THE AFORESAID POLICY AND AGREES TO BE BOUND BY THE EQUAL EMPLOYMENT OPPORTUNITY
CLAUSE 141 CFR 60.101 ISSUED UNDER E.O. 11246 AFFIRMATIVE ACTION CLAUSE 141 CFR 250.41 ISSUED UNDER VETERAN'S ASSISTANCE ACT OF
1974 ON CONTRACTS OR ORDERS OF \$10,000 OR MORE, AND THE AFFIRMATIVE ACTION CLAUSE 141 CFR 741.41 ISSUED UNDER THE REHABILITATION ACT OF
1973, AS AMENDED, ON CONTRACTS OF \$2,500 OR MORE.

Concurrency in C++

P.A. Buhr
G.J. Ditchfield
C.R. Zarnke

Research Report CS-88-30
July 1988

Concurrency in C++

P. A. Buhr*, G. J. Ditchfield*, C. R. Zarnke**

* Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

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

1. Introduction

C++ already supports many programming paradigms: procedural programming, data hiding, data abstraction, and object-oriented programming [Str87]. All of these are subdivisions (though not necessarily disjoint) of the imperative programming style.

A paradigm that is not supported by C++ is *multi-process structuring*, where a program is designed as a set of processes that cooperate to solve a problem [Che82]. This paper considers several ways to add support for concurrency to C++. A number of alternatives must be considered because of the number of programming paradigms available in C++, any of which could be adapted to provide multiple processes.

Any scheme for providing concurrency must provide a way to start new processes, a way to synchronize the execution of processes, and a way for processes to communicate with each other. We also impose the following requirements.

- Both static and dynamic process creation must be supported, that is, processes can be created by declarations or by allocation using the new operation. Both forms of process creation are necessary to make maximum use of concurrency in a particular algorithm and in the hardware resources.
- Static type checking of all communications between processes must be done. We feel that static type checking is extremely important for early detection of errors and efficient generation of code. As well, this requirement is consistent with the fact that C++ is already a statically typed programming language.
- There must be some way to control the duration of synchronization between processes so that there is complete flexibility in the order that a process can respond to requests. This requirement has been shown to be of fundamental importance in concurrency systems, as demonstrated in the send-receive-reply paradigm [Gen81]. Without it, certain classes of concurrency problems are quite difficult to implement, and the amount of concurrency that can be specified by the programmer is restricted.
- Any system chosen should allow concurrency within an address space or in many address spaces, and should support distributed processing.

For the sake of simplicity, this paper will not deal with remote processes in detail, although we have endeavored to keep these problems in mind.

- Any system chosen must blend well with the syntax, semantics, and philosophy of C++.

2. Processes

This section discusses some ways of introducing processes into C++.

2.1 Unix style

The Unix¹ operating system provides concurrency with the `fork()` system call, which creates a new process which is a nearly-exact copy of the parent. The parent and child can identify themselves by examining the value returned by `fork()`.

This form of process creation is unsatisfactory for a number of reasons.

- Since the mechanism for process creation is not part of the programming language or a standard library, programs that use it are not necessarily portable to other operating systems that provide different concurrency primitives.
- Whole program duplication means that there is no clear distinction between the parent process's code and that which defines the action of the new process, and is wasteful of storage.
- Creation of separate address spaces is not always desirable. Running multiple processes in a single address space can have distinct advantages in efficiency when transferring information among the processes.

2.2 Mesa style

In the Mesa style [MMS79], new processes are created by invoking a function in a special way. The process is given its own state, and a new thread of control executes the function body. When the function body returns, the thread ceases execution. Some facility is provided which allows the original process to wait for the new process to terminate, deletes the new process, and returns the value returned by the function body, if any. Information may be passed to the process through the function's parameter list.

This approach is a good extension to the existing "sequential" programming style in C, as the notion that execution of a program is essentially a concurrent invocation of `main(...)` is simply generalized.

Processes of this form can be added to the language in several ways. First, it can be added as a library along the lines described in [Cor88]. Briefly,

```
Process pid;
```

declares that `pid` can hold a process identifier.

```
pid = emit(5000, f, 3, 3.5);
```

creates a process to execute the function `f()` with 5000 bytes of stack space², passes 3 and 3.5 to `f` as arguments, and returns a process identifier.

```
v = absorb(pid);
```

¹Unix is a trademark of AT&T Bell Laboratories

²This is needed if the new process runs in the same address space as the current process.

waits for that process to terminate, and assigns whatever value was returned by `f` to the variable `v`. A second, alternative, which allows static process creation, is to perform the `emit()` and `absorb()` implicitly in the constructor and destructor of the `Process` class.

These schemes result in poor type checking of the arguments and return type of the emitted function, or place severe restrictions on the types of functions that can be emitted, or both. A third alternative regains type checking by extending C++ with a new derived type `process` and two new programming language constructs for starting and synchronizing with the process. The type “process executing *function type*” should be analogous to “pointer to *function type*”. For example,

```
int (process pid)(int, float);
```

declares a variable `pid` that can contain a process identifier for a process that executes a function which takes an integer and a real parameter and returns an integer.

The `process` construct can then be used to emit a function running concurrently with the calling function, for example:

```
int foo(int x, float y);
pid = (process(5000) foo)(3, 3.5);
```

The general form of this call is: `(process(stack-size) function-valued-expression)(argument-list)`. It yields a process identifier as its result.

The process identifier is used in the `join` construct to synchronize with the new process.

```
int result = join pid;
```

All of these variations have the disadvantage that the programmer must be explicitly aware of process identifiers, and nothing in the variations can be construed as object-oriented.

2.3 Concurrent Objects

In this style, C++ is extended with concurrent classes. A call to a member function of such a class is considered to start the member executing concurrently. (Alternately, the function could be executed by a thread of control associated with the concurrent object.) The member function call also provides a means of synchronization, and transfers data between the processes. Languages that take this approach include ConcurrentSmalltalk [YT86] and ABCL/1 [YBS86]. It differs from the Mesa style in that only one member function for a particular concurrent object can be executing at a time. This restriction ensures that processes have mutually exclusive access to the object's members³.

Inheritance among concurrent classes is possible. The member functions of a concurrent class are likely to interact with each other in intricate ways, so the addition or re-writing of member functions will require a thorough understanding of the implementation of the base process.

This approach has the advantage that it is a semantically simple extension to a class based language, requiring few extensions to the language. It requires that the compiler generate special code for member function calls, access to static class members, and access to public members of concurrent classes.

³Mutual exclusion over all instances of a class must also be provided for access to the static members of the class.

2.4 Ada Style

In the Ada⁴ style, new processes are created from instances of class-like constructs [Ada83]. It differs from the Concurrent Object style in that the thread of control executes a special section of code that controls which call to a member function of the class will be processed next, and which automatically provides some concurrency.

This style of process creation requires a new type specifier, `process`, which has all the properties of a class, and which must contain a member called `main()`. When a process object is created, a new thread of control is created which executes the constructor, calls `main()`, executes the destructor for the process type, and waits for the process to be deleted. The process that created the new process continues execution at the point where the new process was created. `main()` has protected visibility, since it can not be called by the user, but it may be redefined by a derived class. Arguments can be passed to `main()` by constructors in the same way that arguments are passed to constructors of base classes, for example:

```
process fred {
    protected:
        main(int i);
    public:
        fred(int i, int j) : main(j) { ... };
}
```

However, the difference is that the constructor is executed first and then `main()` is invoked.

A block will not terminate until all the processes statically declared within it have terminated. This is easy to implement by having the compiler implicitly insert the equivalent of a join operation at the point where the process goes out of scope. Dynamically created processes can be explicitly waited for by performing a delete operation, which will also implicitly perform a join.

In general, there are no problems with inheriting from a process. The derived class inherits the members of the base class. The `main()` member that is executed by the process is the one defined in the derived class. If the derived class does not define its own `main`, then it can not define any new member functions. This is because the `main()` in the base class only controls the calls to the set of functions it has defined.

Multiple inheritance from several processes seems straight-forward. Multiple inheritance from mixes of processes and classes also seems simple, if processes can contain “ordinary” member functions and variables. The result is a process with the members of the ordinary classes, as well.

This scheme is a simple extension of the “object-oriented” approach where an object is viewed as executing a method in response to being sent a message. Constructors and destructors provide convenient initialization and termination of processes. The specification of a block of code to be executed by the process allows flexibility in the handling of requests from other processes. Processes can be allocated statically or dynamically by putting them on the stack or on the free store.

One disadvantage of the scheme is that it conflicts with the idea that a program is a process executing the (ordinary) function `::main()`. However, if `::main()` is viewed as a member of a process class that contains the entire program, the conflict disappears. Another disadvantage is that process can not be implemented as a standard library. One would have to create process types by deriving from a class `Process`. `main()` would be a virtual member that would be spawned by `Process::Process()` using the concurrency primitives of the operating system. But then the derived process’s `main()` would begin execution *before* the derived class’s constructor!

⁴Ada is a trademark of the U.S. Government

3. Synchronization and Communication

3.1 Pipes

In the Unix *fork()* model, synchronization and communication are done by reading and writing from special objects called *pipes*, which provide buffered streams of bytes.

Since pipes are streams of bytes, communication through them is not type-safe. Programmers are forced to build type-safe abstractions over top of them.

3.2 Concurrent Objects

Synchronization in the Concurrent Object scheme is provided by the calls to the member functions. The maximum amount of concurrency is provided by having the caller resume execution immediately without waiting for the called function to do anything, but this makes it difficult for the member function to return any data.

If data is to be returned, the calling process must block until the function returns a value. If this is done by executing a return statement, there is no gain in concurrency. To regain the concurrency, C++ must be extended with a reply statement which specifies a value to be returned to the calling process, allows the caller to resume execution, and allows the called function to continue executing the remainder of the function body.

Member function calls are processed in first-in, first-out order. *Futures* are an additional synchronization mechanism that allows requests to be processed in arbitrary order. *future* is a generic container type with *set* and *receive* operations. Future objects are either empty or contain an object of the parameter type. A concurrent member function that can not fulfill its function immediately returns an empty future. The caller performs a *receive* on the future when it needs the returned value. The *receive* operation blocks until the future is set⁵. When a member function returns an empty future, the future will have to be noted in the object's data structures so that later invocations of member functions can fill it in. This is likely to result in complex interactions between member functions.

3.3 Ada Style

Ada style synchronization (*rendezvous* in Ada) involves synchronous calls to the members (*entries* in Ada) of a process (i.e. the caller blocks until the member function returns with its results), and passing information using the standard argument-parameter mechanism. The process's *main()* member decides which call to which member function will be accepted next. The following are the statements and declarations that are required in a process to support this form of synchronization.

3.3.1 entry Members

Either all member functions will have synchronization associated with their calls (except the constructor and destructor, since no concurrency is possible during their execution), or there must be a way to specify those members that will be synchronized and those that will not. The argument for not having synchronization on all calls is that simple functions that perform reads from the process's member variables can be accomplished asynchronously, hence saving the execution time cost of synchronization. The argument against is that if an asynchronous function changes the state of the process, then the integrity of the process is forfeit. At this point, we feel that providing

⁵This variant of the *future* concept is most like the ConcurrentSmalltalk *CBox*.

both kinds of member functions is reasonable. Distinguishing these two kinds of functions is done by prefixing a member function with the clause `entry`, as in:

```
process fred {  
    ...  
    entry ... foo1(...); // synchronization on call  
    ... foo2(...);      // no synchronization on call  
}
```

`entry` functions can never be `inline` because synchronization code must be in the member not at the call site.

3.3.2 `accept` Statement

The `accept` statement is used to dynamically choose which call to an entry member will be allowed to occur next. When an `accept` is executed, the process instance is `accept-blocked` until a call to that particular entry member occurs. Two forms of the `accept` statement are possible depending on whether the code executed when an entry call is accepted is placed in the `accept` statement or outside of it.

Out-of-Line Accepts In this scheme, a block of code is specified for an entry member just as it is for an ordinary member. An `accept` statement has the form: `accept entry-member-name;`. When an entry call is accepted, the body of the entry member is executed. If the caller is expecting a return value, then this value must be specified in the body of the `accept`. When the body ends, the caller and the acceptor continue execution at the entry call and the `accept` statement, respectively. POOL-T [Ame87] takes this approach, and adds a “post-processing” section to the member body that is executed after the caller has been allowed to resume execution.

If inheritance of process types is supported, virtual entry members might be useful.

In-line Accepts This scheme is like that used in Ada. There is no body associated with an entry member using the usual mechanisms in C++. Instead the body becomes a block of statements that forms part of the `accept` statement, as in: `accept entry-member-name(parameter-list) statement;`. If the `accept` body must return a value, it can not do so with a `return` statement because that would cause the containing function to return. The syntactically similar, `reply expression;` statement is used. When it is executed, the calling process resumes execution at the call point and the called process resumes execution after the `statement` in the `accept`.

In-line `accept` bodies require more language extensions than out-of-line `accept` bodies. Specification of the parameter list is the difficult part in C++ as there is nothing in the language that is roughly equivalent. Since functions can not be nested in C++, there is no precedent for such a facility. Out-of-line bodies bear greater resemblance to ordinary member functions.

Anything that can be done with in-line `accepts` can be done with out-of-line `accepts`. However, in-line bodies are arguably more concise and readable. Out-of-line bodies have to communicate with the code that executes `accept` statements by leaving “memos” in the process’s data structures. In cases where the in-line form has several different `accepts` for the same entry, out-of-line bodies must start with switching logic to determine which case applies. We choose program simplicity over language simplicity and recommend in-line `accept` bodies.

3.3.3 suspend Statement

The Ada rendezvous mechanism restricts the order in which calls can be replied to. If the acceptor does not wish to, or can not deal with the currently accepted message, but wants to continue receiving calls, then new accepts must be nested in the current accept clause. This does not allow a dynamic number of deferred calls, and enforces a last-in, first-out order of processing. What is necessary is the ability to suspend the current accept and requeue the accept so that it can be reaccepted at a more appropriate time.

One way to accomplish this is with a suspend statement, which has the form: `suspend entry-member-name`. `suspend` terminates execution of the accept body at that point, but the caller remains blocked and the caller's request is requeued at the end of the specified entry member's queue. This request can then be reaccepted (by the named entry) at some time in the future when the request can be handled by the accepting process.

3.3.4 Request Queues

Suspending onto entry members does not work, in general, because there is no way to differentiate between newly arrived calls and suspended ones. Without this distinction, it is possible to loop infinitely accepting and suspending the same call. Therefore, it is necessary to have queues internal to the process on which calls can be suspended, called **request queues**.

A request queue must be specified with the type of the entry calls that can be suspended on it, so that its use can be checked by the compiler, for example:

```
int (requestqueue q[5])(int, float);
```

This declares an array of request queues that can have entry calls of the specified type suspended on it. A request queue variable can then appear in an accept statement just like an entry member, as in:

```
accept q[3]; // using out-of-line accept style
accept q[3](i, f) { ... } // using in-line accept style
```

In the out-of-line accept style, it is necessary to assign a function body to the request queue variable to define the code that will be executed when an entry call is accepted, for example:

```
process bar {
    int (requestqueue q[5])(int, float);
    int foo(int i, float f) { ... }
    public:
        bar() { q[0] = &foo; q[1] = &foo; ... }
}
```

It is possible to check an entry/request queue to determine if there are any processes waiting on it using an attribute, as in: `q[3].isEmpty()`.

If a request queue is public or passed to another process, many processes could attempt to accept from one queue at the same time. Because of the complexity in implementing this, we choose to allow only one process to receive from a request queue, but allow multiple processes to suspend on a shared request queue.

3.3.5 select Statement

A `select` statement is provided which has the same semantics as the Ada `select` statement.

```
select {  
    when ( /* conditional-expression */ ) // guard on accept  
        accept ... ;  
    or when ( /* conditional-expression */ )  
        accept ... ;  
    ...  
}
```

The `select` statement lets a process accept an entry call on any of the entries whose guards are true. If no entry calls are waiting, the process waits for a call to be made to one of those entries. (A facility to wake up the process after some period of time should also be provided; however, we have not yet dealt with this.) In the implementation, the accepts are tested in the order that they appear in the `select` list. It is up to the programmer to make sure that requests for a particular entry/request queue are eventually accepted.

3.3.6 Monitor Style Synchronization

The `process` construct and its subsequent instantiation provides the mechanism to start a new thread of control at execution time. Request queue variables and `accept` and `suspend` provide the mechanism for synchronization. An interesting question is whether these two facilities are in fact orthogonal in the design. Clearly, it is possible to start separate processes that do not need to synchronize with other processes because they are not performing operations on shared data. However, is the opposite true? Does it make sense to have a non-process object, such as a class, executing accepts and suspends? In fact, this is largely what happens in a monitor which synchronizes multiple processes as they make calls to member functions of the monitor. Appendix A illustrates this by using a class to create a monitor which implements a bounded buffer. Instances of this synchronization object can then be used for communicating information among process instances. The point here is not to mimic monitors in their entirety, but to show that the synchronization constructs can be used in unorthodox ways.

3.4 Messages

In message passing systems, information to be transmitted is bundled up into a *message*, which is a visible, manipulable object. One process executes a `send` operation to transmit the message, and the other process executes a `receive` operation to pick it up.

The sending process may continue execution after performing the *send*, or it may be blocked until the message is received. A third option is to have the sending process block until a `reply` operation is performed. We prefer this form because the simple semantics and implementation match C++'s minimalist philosophy, because information can be returned conveniently to the sender via the `reply`, and because the `reply` operation allows flexibility in the order in which requests are processed.

In traditional send/receive/reply systems, messages and replies are sent to processes. This interferes with type checking, since a process may have to receive more than one type of message or expect more than one type of reply. Instead, messages should be sent to *message queues*, which are objects whose type includes a message type and a reply type. Processes then have message queues

for each message type they handle. A consequence is that some means of receiving from any of a set of message queues is needed.

The implementation of message queues becomes more complex if many processes can attempt to receive from one queue at the same time. As for request queues, we choose that each message queue is owned by one process, which has the exclusive right to receive from it.

The system described above can be described using the existing class construct and the preprocessor “generic” tools. The send and receive operations can be member functions of a generic `MessageQueue` type with message type and reply type parameters, and `reply` is a member function of a matching `Message` type which is used by receiving processes. One approach to waiting on a set of queues is to have a `waitFor` function that takes a list of message queue arguments, waits for a message to arrive on one, and returns a code indicating which queue contains a message.

A more novel approach is to use *access types* [BZ86]. A generic access class is defined for each message queue class. Receive operations are done implicitly by creating an instance of the access class, and `reply` is done implicitly by deleting the instance. The message is accessed through the access variable. This scheme has the advantage that the scope of an automatic access variable statically links the reception of and the reply to a message, which is sufficient for most programs.

A third approach is to add message and message queue types, `send`, `receive` and `reply` operations, and a switch-like `select` statement to the language. The compiler can then statically check that the process receiving messages from a queue is the owner of the queue.

4. Conclusions

We consider Ada-style processes with in-line accepts and request queues to be best of these alternatives for several reasons.

- It does not add explicit process identifiers to the language, as does the Mesa-style approach. Instead, the existing concept of “object” is used.
- Request queues let processes service entry calls in any order without added protocols or internal data structures.
- The acceptance of a request and the reply to the requestor are tied together syntactically; in this case, the reply is implicit in the end of the accept body.
- An implementation of entry calls can choose to leave the arguments of the call in the caller’s stack in circumstances where other styles, such as message passing, must copy them into the called process.

Currently, we have finished the basic design for the Ada-style (out-of-line) style, and we are experimenting with implementations based on a light-weight concurrency kernel written in C [Cor88]. At the moment, the new statements are translated by a simple preprocessor (and by hand in places). By October (the conference), we should have completed a preprocessor based on the g++ parser to perform the transformation automatically to the C code level. If time permits, we then intend to augment the g++ compiler to generate code directly.

5. Comparison with Other Work

There are large number of concurrency designs in an equally large number of programming languages. We have selected only programming languages that provide static type checking of communicated data.

5.1 Ada

The major deficiency with the Ada implementation of concurrency is that the servicing of requests can not be postponed because of the lack of request queues. However, there is a work-around which involves designing protocols with multiple entry calls. A minor deficiency is that entries can not return values directly, but must use the argument-parameter mechanism.

5.2 BNR Pascal

BNR Pascal [GKC87] provides an Ada-like rendezvous along with a type `QUEUE`, and `DEFER` and `REENTER` statements. Unlike our request queue variables, an instance of `QUEUE` allows any type of entry call to be suspended on it. Suspension is done with `DEFER(queue-name)` which suspends the current process on the specified queue. Re-acceptance is done with `REENTER(queue-name)` which takes the process at the head of the queue (if any) and places it at the front of the entry on which it initially arrived. Hence, a process can not be suspended on an entry other than the one on which it initially arrived, which allows the compiler to maintain type safety. This scheme results in switching logic at the start of accept bodies to determine whether a call has resulted from a `REENTER`, and why it was deferred.

5.3 SR

SR [Aea88] provides concurrent objects that can have both concurrent member functions and entry members with in-line accept bodies. The way in which a member is invoked (by a `call` or `send`) indicates whether the caller will block 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.

We are not convinced that all 4 combinations are useful. Some combinations, such as multiple sends to the same object, can be extremely error prone unless the members are carefully written to deal properly with it. Our contention is that, in general, entities are written assuming that they will be invoked in a particular way (i.e. sequentially, alternating (coroutining), or concurrently) and will not work correctly if used in another way. To this end, SR provides pragmas that indicate that a particular member can only be invoked in a particular way. Rather than take this general, and potentially dangerous, approach of allowing the particular invoking form to specify the semantics of execution, we have the definition (`class` or `process`) indicate this essential semantic information and use a uniform syntax for invoking the members.

SR synchronization primitives are essentially the same as Ada. Therefore, it can not respond to messages in arbitrary order. No mechanism exists to suspend a received call, accept new calls, and then re-accept the suspended call at a later time.

5.4 BETA

BETA [KMMN87] is like SR in that the way in which a member is invoked (by a concurrent imperative or alternating imperative) indicates whether it will execute concurrently or not. Hence, the caller, and not the definition, has control of this important aspect of a definition's behaviour.

As well, BETA's synchronization primitives are essentially the same as in Ada. Therefore, it can not respond to messages in arbitrary order.

References

- [Ada83] *The Programming Language Ada: Reference Manual*. United States Department of Defense, February 1983.
- [Aea88] Gregory R. Andrews and Ronald A. Olsson et. al. An overview of the SR language and implementation. *Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
- [Ame87] Pierre America. POOL-T: a parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220, The MIT Press, 1987.
- [BZ86] P. A. Buhr and C. R. Zarnke. A design for integration of files into a strongly typed programming language. In *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, pages 190–200, October 1986.
- [Che82] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier, 1982.
- [Cor88] G. V. Cormack. A micro kernel for concurrency in C. *Software-Practice and Experience*, 18(4):485–491, May 1988.
- [Gen81] W. Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software-Practice and Experience*, 11:435–466, 1981.
- [GKC87] N. D. Gammage, R. F. Kamel, and L. M. Casey. Remote rendezvous. *Software-Practice and Experience*, 17(10):741–755, October 1987.
- [KMMN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48, The MIT Press, 1987.
- [MMS79] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual*. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.
- [Str87] Bjarne Stroustrup. What is “object-oriented programming”? In *Proceedings of the First European Conference on Object Oriented Programming*, June 1987.
- [YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPLSA '86*, pages 258–268, November 1986. Special issue of SIGPLAN Notices, vol. 21, no. 11.
- [YT86] Yasuhito Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *OOPLSA '86*, pages 331–340, November 1986. Special issue of SIGPLAN Notices, vol. 21, no. 11.

6. Appendix A — Monitor Style Synchronization

This style of synchronization object works like a Hoare monitor because the acceptor (signaller) blocks while the function associated with the request executes to completion. `accept` is not the same as a monitor `signal` because an `accept` on an empty entry/request queue blocks, while a `signal` on an empty condition does nothing and execution continues. Therefore, it may be necessary to check explicitly whether there are processes waiting on an entry/request queue before performing an `accept` to prevent receive-blocking that would lead to deadlock. As well, `suspend` is not the same as `wait` because resumption of a suspended function restarts the function while waiting is restarted after the `wait` statement.

```
class BoundedBuffer {
    const int Size = 100;
    int front, back;           // position of front and back of queue
    int Elements[Size + 1];    // queue of integers
    void (requestqueue Full)(int elem); // wait if queue is full
    int  (requestqueue Empty)();   // wait if queue is empty
public:
    BoundedBuffer() {
        front = 0;
        back = 1;
        Full = &insert;
        Empty = &remove;
    }; // BoundedBuffer

    entry void insert(int elem) {
        if (front == back) suspend Full;
        Elements[back] = elem;
        back = (back + 1) % (Size + 1);
        if (!Empty.isEmpty()) accept Empty;
    }; // insert

    entry int remove() {
        int elem;

        if ((front + 1) % (Size + 1) == back) suspend Empty;
        front = (front + 1) % (Size + 1);
        elem = Elements[front];
        if (!Full.isEmpty()) accept Full;
        return(elem);
    }; // remove
}; // BoundedBuffer

process Producer {
    Producer(BoundedBuffer buf) {
        for ( ... ) {
            item = ... // produce item
            buf.insert(item);
        }
    }
}
```

```

        }
    } // Producer
} // Producer

process Consumer {
    Consumer(BoundedBuffer buf) {
        for ( ... ) {
            item = buf.remove();
            // consume item
        }
    } // Consumer
} // Consumer

main() {
    BoundedBuffer buf;           // create a communication buffer
    Producer Prod1(buf), Prod2(buf); // create producers
    Consumer Cons(buf);          // create consumer
} // main                       // wait for process completion

```


7. Appendix B — Disk Scheduler

The following example illustrates a fully implemented disk scheduler using the Ada style (out-of-line) concurrency extensions to C++. It demonstrates two facilities that are not available in Ada. First, suspension of an accept body without unblocking the caller, and second, request-queue variables. The disk scheduling algorithm used in the example 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. (Ada can only support arrays (families) of entries which is expensive in both search time and storage utilization.) 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 will be serviced on that scan.

To prevent deadlocks between the disk and server, the disk calls the server to get the next request that it will service. This call does two things: it passes to the server the status of the just completed disk request which is then returned from server to client, and it returns the information for the next disk operation. To prevent the server from having to make a local copy of each caller's request information when it cannot be serviced immediately, another request queue is introduced. When accepted, the code for this queue copies the parameter values from the caller's parameters to local variables in the server and then suspends on the request queue `CurrentReq`. Hence, the server only has a single copy of the request that is currently being serviced by the disk. The cost is the retrieval of the values from the caller's stack.

```
enum logical { FALSE = 0, TRUE = 1 };

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

enum IOStatus { INITIAL, COMPLETE, ERROR, EOF };

class IORequest {
public:
    int cylinder;
    int sector;
    BufferAddress *bufadr;
}; // IORequest

process Disk {
    logical Alive;
    IOStatus status;
    IORequest WorkRequest;
protected:
    main(Server &);
public:
    Disk(Server &server) : main(server) {};
}; // Disk

Disk::main(Server &server) {
    Alive = TRUE;
```

```

    status = INITIAL;

    for (;Alive;) {
        WorkRequest = server.WorkRequest(status);
        status = COMPLETE;
    } // for
} // Disk

process Server;                                // forward declaration

class WaitingRequest : public Sequable {
public:
    int cylinder;
    IOStatus (Server::requestqueue req)(IORequest);
}; // WaitingRequest

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

process Server {
    Sequence(WaitingRequest) Requests; // list of client requests
    WaitingRequest *Current;
    logical Alive, DiskInUse, Direction;
    IORequest CurrentWork;
    IOStatus CurrentStatus;

    IOStatus (requestqueue ServeRequest)(IORequest &);
    IOStatus (requestqueue CurrentRequest)(IORequest &);
    IORequest (&requestqueue DiskWaiting)(IOStatus);
protected:
    main();
public:
    entry IORequest WorkRequest(IOStatus);
    entry IOStatus DiskRequest(IORequest &);
    entry void Die();
}; // Server

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

    Alive = DiskInUse = Direction = TRUE;

    for (;Alive;) {
        select {                                     // in order of importance
            accept Die;                               // request from system
            accept WorkRequest;                       // request from disk
            accept DiskRequest;                       // request from clients
        } // select
    }
}

```

```

    } // for
} // Server

IORequest Server::WorkRequest(IOStatus status) {
    if (status != INITIAL) { // 1st time is a special case
        CurrentStatus = status;
        accept CurrentRequest; // reply to waiting client
        // advance to the next disk request in the current direction
        WaitingRequest *temp = Current;
        Current = Direction ? Requests.succ(Current): Requests.pred(Current);
        Requests.remove(temp); // remove just processed request
        delete temp;
        if (Current == 0) { // reverse direction ?
            Direction = !Direction;
            Current = Direction ? Requests.head(): Requests.tail();
        } // if
    } // if

    DiskInUse = FALSE;
    if (!Requests.isEmpty()) { // any clients waiting ?
        DiskInUse = TRUE;
        accept Current->req; // get work from waiting client
        // the global variable CurrentWork is assigned the current request
        return(CurrentWork); // return work for disk
    } else {
        suspend DiskWaiting; // wait for client to arrive
    } // if
} // WorkRequest

IOStatus Server::DiskRequest(IORequest &req) {
    if (DiskInUse) {
        // insert into list by ascending order of cylinder number
        for (WaitingRequest *lp = Requests.head();
            lp != 0 && lp->cylinder < req.cylinder;
            lp = Requests.succ(lp));
        WaitingRequest *np = new WaitingRequest;
        np->cylinder = req.cylinder;
        np->req.Rtn = &Server::ServeRequest;
        if (Requests.isEmpty()) Current = np; // 1st client, so set Current
        Requests.insert(np, lp);
        suspend np->req; // suspend until request is to be serviced
    } else {
        DiskInUse = TRUE;
        CurrentWork = req;
        accept DiskWaiting;
        suspend CurrentRequest;
    } // if
}

```

```

} // DiskRequest

void Server::Die() {
    Alive = FALSE;
} // Die

IOStatus Server::ServeRequest(IORequest req) {
    CurrentWork = req;
    suspend CurrentRequest;
} // ServeRequest

IOStatus Server::CurrentRequest(IORequest req) {
    return(CurrentStatus);
} // CurrentRequest

IORequest Server::DiskWaiting(IOStatus status) {
    return(CurrentWork);
} // DiskWaiting

process Client {
    protected:
        main(Server &, IORequest &);
    public:
        Client(Server &server, IORequest &req) : main(server, req) {};
} // Client

Client::main(Server &server, IORequest &req) {
    IOStatus status;

    status = server.DiskRequest(req);
} // Client

const int NoOfTests = 10;

IORequest test[NoOfTests] = { { 20, 0, 0 },
    { 99, 0, 0 }, { 0, 0, 0 }, { 15, 0, 0 },
    { 30, 0, 0 }, { 4, 0, 0 }, { 16, 0, 0 },
    { 80, 0, 0 }, { 85, 0, 0 }, { 2, 0, 0 } };

main() {
    Server server;                // start the disk server
    Client *p[NoOfTests];
    int i;

    for (i = 0; i < NoOfTests; i += 1) {
        p[i] = new Client(server, test[i]); // start the clients
    } // for

```

```
for (i = 0; i < NoOfTests; i += 1) {  
    delete p[i];          // wait for completion of the clients  
} // for  
  
Server.Die();             // terminate the disk server  
} // main
```



Computer Resources International A/S

MAY 26 1989

University of Waterloo
Department of Computer Science
Waterloo, Ontario N2L 3G1
Canada

1989-05-22

Our ref.

Your ref.

Project

Dear Sirs,

Please send us the reports indicated ~~on the attached~~
~~copy of your report~~ below.

Thanking you in advance, we remain

Yours sincerely,
Computer Resources International A/S

Anne Marie Larsen
Anne Marie Larsen

Reports wanted: CS-88-31
CS-88-25
CS-88-30

*sent reports
May 26/89*

Volker Hüsken
Lehrstuhl für Betriebssysteme
RWTH Aachen
Prof. Dr. D. Haupt

Kopernikusstr. 16
D-5100 Aachen
West-Germany
bitnet: huesken@dacth01.bitnet

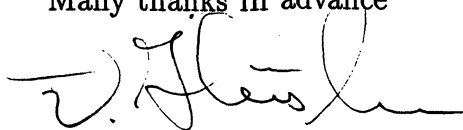
P. A. Buhr
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1

Aachen, 12. April 1989

Dear Mr. Buhr,

I just read your very interesting article on "Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language" SIGPLAN NOTICES April 1989. One of your references is the **Research Report CS-88-30 "Concurrency in C++"**, which you hopefully could send me to the above address. We are working in the same area of parallelizing C++ and we have finished a first attempt, which has a different granularity than your model. This work is written in German so I don't know whether this is of interest to you.

Many thanks in advance



V. Hüsken

*Sent 88-30
May 12/89*

PHONE CALL

Date 16 Mar 89 Time 10³⁰

To Sue DeAngelis

WHILE YOU WERE OUT

M David Bern

of _____

Phone (201) 6681593

Telephoned ☒ Please call ☐

Called to see you ☐ Will call again ☐

Wants to see you ☐ Returned your call ☐

MESSAGE P.O. Box 4186

Warren, NJ 07060

1 copy of

CS-88-30 Buhr/Witchfield

+ listing + Zarnke

of available reports sent March 17/89

Operator U.

031020

today's

PHONE CALL

Date March 14 Time 4:10

To Sue DeAngelis

WHILE YOU WERE OUT

M David Bess

of New Jersey

Phone 201-668-1593

Telephoned ☒ Please call ☒

Called to see you ☐ Will call again ☐

Wants to see you ☐ Returned your call ☐

MESSAGE _____

re: report CS 88/30
by Ditchfield + ?

Operator Pat

031020

today's

MARTIN MARIETTA ENERGY SYSTEMS, INC.

REMITTANCE STATEMENT

(PLEASE NEGOTIATE PROMPTLY)

DATE

CHECK

PACKING LIST/ INVOICE NO.	OUR ORDER	REL. NO.	MATERIAL OR SERVICE	TRANSPORTATION	DISCOUNT	DEDUCTION	CD. *	NET
------------------------------	-----------	-------------	---------------------	----------------	----------	-----------	-------	-----

02XHV751

2.00

2.00

*Cheque was made
out to Geography Dept
Cashiers office told
me to just deposit
in our account as is
as we are going to
lose money on it
anyway.
mailed report
Nov. 22/88
S.D.*

*** DEDUCTION CODE
EXPLANATION:**

- | | | | |
|--------------------------------------|---|--|--------------------------------|
| 1. Unauthorized Ins. or Value Charge | 4. Furnish invoice for transp. and/or supporting copy of freight bill | 6. Rejected item or unacceptable overage | 8. Contract retention |
| 2. Transportation for your account | 5. Unauthorized price increase | 7. Taxes not applicable | 9. Debit/Credit memo processed |
| 3. Unauthorized prem. transp. cost | | | UCN-679C (6 4-88) |

Purchase Order No: 02X-HY751V
Issue Date: 11/08/88

ACTING UNDER U.S. GOVERNMENT CONTRACT DE-AC05-84OR21400 WITH THE U.S. D.O.E.

PAGE 1

Seller (U00299)

Refer Questions To:

UNIVERSITY OF WATERLOO
DEPARTMENT OF GEOGRAPHY
ISIAH BOWMAN BLDG
WATERLOO ONTARIO CANADA N2L3G1

MT UNDERWOOD
MARTIN MARIETTA ENERGY SYSTEMS INC
P. O. BOX 2008, MS 6286
OAK RIDGE, TN 37831-6286
(615) 574-0665

Ship To:

Furnish Original Invoice To:

MARTIN MARIETTA ENERGY SYSTEMS, INC.
ACCOUNTS PAYABLE
P. O. BOX 2004
OAK RIDGE, TN 37831-2004

Attention: PO 02X-HY751V

Shipping Point:

Shipping Method:

Transportation Terms:

Payment Terms: A Rem Att FOB Code:

* Seller shall show the Martin Marietta Energy Systems Purchase Order Number and Plant *
* (02X-HY751V AND) on all Packages, B/L and Freight Bills, and Invoices *

Furnish the following items in accordance with Terms and Conditions designated B (06/81) C/S (6/86) Attached hereto or incorporated herein by reference with specifications and/or drawings referred to herein and made part hereof

Item	Quantity	Unit	Unit Price	Total Price	Deliver By
001 ACCT	1	EA	\$2.00	\$2.00	12/07/88

Description

CS-88-30 - CONCURRENCY IN C ++. BUHR, P. S. ET AL.

SHIP TO: MARTIN MARIETTA ENERGY SYSTEMS, INC.
MARTHA UNDERWOOD, 4500N, I-103
P. O. BOX 2008, BETHEL VALLEY ROAD
OAK RIDGE, TN 37831

1 Total Price
1
1 \$2.00
1
1

PRIORITY: DO E-2 This is a rated order certified for national defense use, and you are required to follow all the provisions of the Defense Priorities and Allocations System Regulations (15 CFR Part 350).

1	Martin Marietta
1	Energy Systems, Inc.
1	Purchasing Signature
1	
1	
1	

Purchase Order No: 02X-HY751V
Issue Date: 11/08/88

ACTING UNDER U.S. GOVERNMENT CONTRACT DE-AC05-84OR21400 WITH THE U.S. D.O.E.

PAGE 2

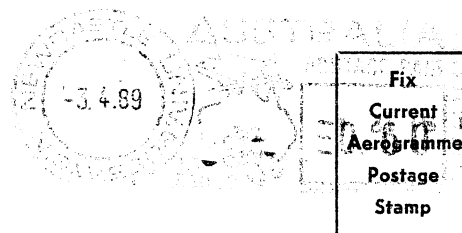
=====

Last Page 2 of 2

AEROGRAMME

BY AIR MAIL

PAR AVION



Research Report Secretary

Department of Computer Science

University of Waterloo

Waterloo

ONTARIO N2L 3G1

C A N A D A

(COUNTRY OF DESTINATION)

Approved by the Australia Post for
acceptance as Aerogramme No. 58

SENDERS NAME AND ADDRESS

Steward

Electrical Engineering and Computer Science

UNIVERSITY OF NEWCASTLE
NEW SOUTH WALES
AUSTRALIA. 2308

POSTCODE

SECOND FOLD HERE

FIRST FOLD HERE

SECOND FOLD HERE

FIRST FOLD HERE



THE UNIVERSITY OF NEWCASTLE

NEW SOUTH WALES, 2308

AUSTRALIA.

3 April 1989

Research Report Secretary
Department of Computer Science
University of Waterloo
Waterloo
Ontario N2L 3G1
CANADA

Dear Sir,

Would you please let us have a copy of the following report:

REPORT

NO.

CS-88-30

TITLE

Concurrency in C ++

AUTHORS

P A Buhr
G J Ditchfield
C R Zarnke

Thank you.

Yours faithfully,

D C Edwards

D. C. Edwards (Mrs)
Secretary
Computer Science

*sent
April 25/89*

To

From

Date

memo

send April 6/89 University of Waterloo
along with 3 papers from Peter Also gave Peter #
88-30 his name + phone

check
Peter Buhler in C++

David Bern
doing C++ training
P.O. Box 4186

Warren, NJ

07060-0186

U.S.A. (201)-668-
1593

write courses in
C++ and does

training



The University of Western Ontario

Dr T. Muldner, Visiting Prof.
Department of Computer Science
Middlesex College
London, Canada
N6A 5B7

London, Jan 18, 89

Would you please send me a copy of the research
report CS-88-30 : Concurrency in C++.
(I enclose \$2.00). Would you also include
my name on your mailing list.

Yours sincerely

Dr T. Muldner

sent
Jan. 30/89

CS-88-29 - PORTABLE COMPUTERS AND DISTANCE EDUCATION

ABSTRACT:

Experiments are being conducted at the University of Waterloo with an integrated computer and communications system for students in distance-education programs. Distance students will be able to use the data communications capability of the telephone system in conjunction with microcomputers to communicate with their teachers on the Waterloo campus and their fellow students.

This system should also allow distance students to have immediate access to teaching materials such as lecture notes, laboratory sessions, and assignments, and to many of the computer-based tools and information sources which are commonly available to on-campus students. Electronic submission and return of assignments should also be possible.

With this new ability to use computers and communications, students enrolled in distance education programmes will have many of the advantages of on-campus students. Microcomputers will not be a substitute for the teacher, but will act as a useful learning tool and facilitate communication with on-campus teachers and other distance students, and provide access to many of the accumulated resources of the university.

AUTHORS: J.P. Black, D.D. Cowan, V.A. Dyck, S.L. Fenton

C.K. Knapper, T.M. Stepien

PRICE: \$2.00

CS-88-30 - CONCURRENCY IN C ++

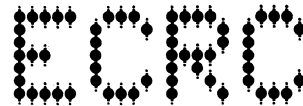
ABSTRACT:

C ++ already supports many programming paradigms: procedural programming, data hiding, data abstraction, and object-oriented programming. All of these are subdivisions (though not necessarily disjoint) of the imperative programming style. A paradigm that is not supported by C ++ is *multi-process structuring*, where a program is designed as a set of processes that cooperate to solve a problem. This paper considers several ways to add support for concurrency to C ++. A number of alternative concurrency models are considered because of the number of programming paradigms available in C ++, any of which could be adapted to provide multiple processes. A type safe model is suggested that is similar to that in Ada with the extension that a process can respond to requests in arbitrary order making it as powerful as the send/receive/reply model. As well, the model continues to support object oriented facilities like subtyping and inheritance.

AUTHORS: P.A. Buhr, G.J. Ditchfield, C.R. Zarnke

PRICE: \$2.00

*sent to U of Hawaii
Jan. 6/89*



EUROPEAN COMPUTER-INDUSTRY
RESEARCH CENTRE GMBH
(FORSCHUNGSZENTRUM)

ECRC GMBH · ARABELLASTRASSE 17 · D-8000 MÜNCHEN 81

University of Waterloo
Dept. of Computer Science
Attn.: Research Report Secretary
Waterloo, Ontario N2L 3G1

ARABELLASTRASSE 17
D-8000 MÜNCHEN 81

☎ 089/92699-0

Ext. 92699-
Nst.

☎ 5216910 Fax 089/92699-170

CANADA

YOUR REF.
Ihre Zeichen

YOUR LETTER OF
Ihr Schreiben vom

OUR REF.
Unsere Zeichen
HG/am

DATE
Datum
14.Nov.1988

Dear Sirs,

Order of Technical Report

We would like to order one paper copy of the following technical report:

* CS-88-30
Concurrency in C++
Authors: Buhr, Ditchfield, Zarnke

We look forward to receiving this publication as soon as possible and thank you in advance for your attention in this matter.

We know that all orders must be prepaid but as we can not draw a cheque on such a small amount we would like to suggest to effect payment cash on receipt of goods.

Yours faithfully,

Astrid Märkl

Astrid Märkl

*sent
with compliments
Nov. 30*

CS-88-33 - ON EFFICIENT ENTREEINGS

ABSTRACT:

A *data encoding* is a formal model of how a logical data structure is mapped into or represented in a physical storage structure. Both structures are complete trees in this paper, and we encode the logical or guest tree in the leaves of the physical or host tree giving a restricted class of encodings called *entreeings*. The *cost* of an entreeing is the total amount that the edges of the guest tree are stretched or dilated when they are replaced by shortest paths in the host tree. We are particularly interested in the *asymptotic average cost* of families of similar entreeings.

Our investigation is a continuation of the study initiated in [6].

AUTHORS: Paul S. Amerins, Ricardo A. Baeza-Yates, Derick Wood

PRICE: \$2.00

CS-88-34 - THE SUBSEQUENCE GRAPH OF A TEXT

ABSTRACT:

We define the directed acyclic subsequence graph of a text as the smallest deterministic partial finite automaton that recognizes all possible subsequences of that text. We define the size of the automaton as the size of the transition function and not the number of states. We show that it is possible to build this automaton using $O(n \log n)$ time and space for a text of size n . We extend this construction to the case of multiple strings obtaining a $O(n^2 \log n)$ time and $O(n^2)$ space algorithm, where n is the size of the set of strings. For the later case, we discuss its application to the longest common subsequence problem improving previous solutions.

AUTHOR: Ricardo A. Baeza-Yates

PRICE: \$2.00

If you would like to order any reports please forward your order, along with a cheque or international bank draft payable to the Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, to the Research Report Secretary.

Please indicate your current mailing address and if you wish to remain on our mailing list.

MAILING ADDRESS:

CTRL
WANG INSTITUTE OF BOSTON UNIVERSITY
72 TYNG ROAD
TYNGSBORO, MA 01879-2099 USA

YES, REMAIN ON MAILING LIST

NO, DELETE FROM MAILING LIST

PLEASE SEND : (1) CS-88-30 CONCURRENCY IN C++

rec'd payment
& sent report

NOV 7 1988

Printing Requisition / Graphic Services

15092

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION

CS-82-30

DATE REQUISITIONED

July 18/88

DATE REQUIRED

ASAP

ACCOUNT NO.

112616051141

REQUISITIONER - PRINT

Sue DeAngelis

PHONE

2192

SIGNING AUTHORITY

Kim Shogren

MAILING INFO -

NAME

Sue DeAngelis

DEPT.

CS

BLDG. & ROOM NO.

DC 2314

☒ DELIVER

☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES 19 NUMBER OF COPIES 100

TYPE OF PAPER STOCK

☒ BOND ☐ NCR ☐ PT. ☐ COVER ☐ BRISTOL ☐ SUPPLIED ☐

PAPER SIZE

☒ 8 1/2 x 11 ☐ 8 1/2 x 14 ☐ 11 x 17 ☐

PAPER COLOUR

☒ WHITE ☐ ☐ BLACK ☐

PRINTING

☐ 1 SIDE ☒ 2 SIDES ☐ PGS. FROM TO

BINDING/FINISHING

☒ COLLATING ☒ STAPLING ☐ PUNCHED ☐ PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

Math Fronts & Backs
Unclosed.

COPY CENTRE

OPER. NO. BLDG. NO. MACH. NO.

DESIGN & PASTE-UP

OPER. NO. TIME LABOUR CODE
D 0 1
D 0 1
D 0 1

TYPESETTING

QUANTITY

P A P 0 0 0 0 0 T 0 1
P A P 0 0 0 0 0 T 0 1
P A P 0 0 0 0 0 T 0 1

PROOF

P R F
P R F
P R F

NEGATIVES

QUANTITY

OPER. NO.

TIME

LABOUR CODE

F L M C 0 1
F L M C 0 1
F L M C 0 1
F L M C 0 1
F L M C 0 1

PMT

P M T C 0 1
P M T C 0 1
P M T C 0 1

PLATES

P L T P 0 1
P L T P 0 1
P L T P 0 1

STOCK

0 0 1
0 0 1
0 0 1
0 0 1

BINDERY

R N G B 0 1
R N G B 0 1
R N G B 0 1
M I S 0 0 0 0 0 B 0 1

OUTSIDE SERVICES

\$ COST

TAXES - PROVINCIAL ☐ FEDERAL ☐ GRAPHIC SERV. OCT. 85 482-2