Concurrent C*

N. H. Gehani
W. D. Roome

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Concurrent programming is becoming increasingly important because multicomputer architectures, particularly networks of microprocessors, are rapidly becoming attractive alternatives to traditional maxicomputers. Concurrent programming is important for many reasons [HOAR78, GEHA84b]:

- Concurrent programming facilities are notationally convenient and conceptually elegant when used for writing systems in which many events occur concurrently, for example, in operating systems, real-time systems and database systems.
- Inherently concurrent algorithms are best expressed when the concurrency is stated explicitly; otherwise, the structure of the algorithm may be lost.
- Concurrent programming can reduce program execution time on genuine multiprocessing hardware such as a network of microprocessors. Efficient utilization of such architectures requires concurrent programming.
- Concurrent programming can reduce program execution time even on single CPU machines because lengthy input/output operations and the CPU operation can proceed in parallel.

The C programming language does not have facilities for concurrent programming. We have designed an upward-compatible extension of C, called Concurrent C, that provides concurrent programming facilities. We chose C because it is a small language that can be implemented efficiently (even on microcomputers). Use of the C language has spread rapidly in the last few years; moreover, it is the language of choice at AT&T Bell Laboratories. Concurrent C is not based on some new concurrent programming model; instead, it is based on a well known model that has been discussed in detail in the literature.

We also have some specific objectives in enhancing C with concurrent programming facilities:

1. To provide a concurrent programming language that can be used for writing systems on genuinely parallel hardware, such as a network of microprocessors or work stations.
2. To provide a test bed for experimenting with a variety of high-level concurrent programming facilities and a distributed programming environment.

In this paper, we discuss the motivation for extending C with concurrent programming facilities and the selection of the concurrent programming model. After this we describe Concurrent C. Then we contrast the concurrent programming facilities in Concurrent C with those in the Ada language; the concurrent programming facilities in both the languages are based on the same concurrent programming model. This is followed by a short discussion of the problem arising from calling C library functions that update shared data. Finally, based on our experience with the first implementation of Concurrent C, we discuss some possible extensions to Concurrent C.

In a companion paper [GEHA84a], we illustrate the use of Concurrent to write a variety of programs.

2. Selection of the Concurrency Model

Concurrent programming models fall into three categories:

1. those based on shared memory,
2. those based on message passing (no shared memory), and
3. a combination of the above two models [ANDR81, ANDR82].

We would like to provide high-level concurrent programming facilities that would work on a *multicomputer* architecture. In a multicomputer architecture, the computers do not share memory. This rules out concurrent programming models that require shared memory (e.g., monitors) for effecient

implementation, and leaves us with models based on message passing.

Message passing models fall into two categories:

1. synchronous message passing (message passing with blocking).
2. asynchronous message passing (message passing without blocking).

We picked the synchronous message passing model because it leads to programs that are easier to understand and implement as compared to the asynchronous message passing model. For example, the asynchronous message passing model requires the presence of underlying message buffers and a sizable message controller [GENT81, DOD79].

In situations where asynchronous message passing is preferable, it can be easily implemented by using an intermediate buffer process.

Synchronous message passing primitives combine process synchronization with information transfer. Two processes interact first by synchronizing, then by exchanging information, and finally by continuing their individual activities. This synchronization is called a *rendezvous.*

In a *simple* rendezvous, the exchange of information is unidirectional—from the message sender to the receiver. A common process interaction idiom involves bidirectional transfer of information—a client process requests service from a server process. The client supplies information to the server about the desired service; the server performs the service and provides the client with the appropriate information. Bidirectional transfer of information requires two simple rendezvous: one to request a service, and another to get the server's reply. The need for two rendezvous imposes additional overhead. Moreover, the client must wait for a reply after making a request. The server may become blocked if the client fails to follow this protocol.

The *extended rendezvous* or *transaction* concept allows bidirectional information transfer using only one rendezvous [DOD79, DOD83]. After a rendezvous has been established, the process requesting service is automatically forced to wait until the server completes the requested transaction. The results of the request are sent back to the waiting client, so that information transfer is bidirectional. From the client's viewpoint, a request for a service, i.e., a request for establishing a rendezvous, is just like a function call.

Although Concurrent C and Ada are based on the same rendezvous model, there are important differences between the concurrent programming facilities in the two languages. These differences are summarized in Section 5.

3. Concurrent C

In designing these concurrent programming facilities, we have tried to make them as orthogonal as possible to the sequential programming facilities. Our goal is to make Concurrent C upward compatible with sequential C.

3.1 Processes and Process Types

First, some terminology. A *process type* is a process template. A *process* is an instantiation of a process type. A process has its own flow-of-control; it executes in parallel with other processes. Declaring a process type does not automatically create a process of that type. Instead, the programmer must explicitly create each process at run-time. Every process is of some process type, and several distinct processes can have the same process type.

One can think of each process as having its own stack, machine registers, program counter, etc. There is an underlying scheduler that runs these processes on the available processors.[1]

1. A scheduler is not necessary for processors that are dedicated to executing a single process.

A process type has two parts: a *specification* and a *body*. The process type specification (or process specification, for short) is the public part of a process type. Only the information specified in the process specification is visible to other processes. A process specification gives all the information necessary for interacting with a process of that type, or for creating a new process of that type. A process body contains the code (and associated declarations and definitions) that is executed by a process of this type; it is analogous to a function body, which it resembles. Details of the process body are not visible to other processes. (The process specification and body are described in detail in Sections 4.2 and 4.3.)

In any Concurrent C program, the process specification for a process type must precede all references to that process type,[2] and must precede that process's body. The process body must exist in some file that is loaded into the program, but other processes do not need to see the body. In short, a process specification is similar to a declaration of a function's return type and arguments, and a process body is equivalent to that function's body. A common technique is to put the process specification in one file, and the body in a separate file. All separately compiled files containing transaction calls to a process include the file containing its process specification; the file containing the process body also includes file with the process specification. The file with the process body can be compiled separately.

3.2 Process Specifications and Transaction Declarations

A process specification has the form[3]

```
process spec process-type-name(process-param-declarations);
```

or

```
process spec process-type-name(process-param-declarations)
{
  transaction declarations
};
```

The above process specifications declare a process type

```
process process-type-name
```

A transaction declaration has the form

```
trans return-type t-name1(formal-param-declarations), ..., t-namen(formal-param-declarations);
```

where *return-type* is the type of the value returned by the transactions and $t\text{-}name_i$ ($1 \le i \le n$) are the names of the transaction.

Instead of a process parameter or transaction formal parameter declaration, the user can alternatively supply just the formal parameter type in the process specification and transaction declaration.

Concurrent C processes communicate primarily by means of transactions (as discussed earlier in Section 3). The process containing a declaration for a transaction named T may be thought of as a server for transaction T because it will execute requests for this transaction on behalf of the client processes. (Note that a process may act as a server for more than one transaction). When a client process wants service, it says ''I want server process S to execute transaction T for me.'' Now if the server S is not waiting to execute a transaction request then the client must wait until the server says ''I am ready to accept a transaction request.'' Similarly, a server process must wait when it says it is ready to execute a transaction but there is no outstanding request.

---

2. There is one exception to this rule—if specification of process type X refers to process type Y, then the specification of process type X is allowed to precede the specification of process type Y. This exception allows circularity in process specifications.

3. CHANGE/NEW Formal parameter names optional (MP and RFC would prefer that); RFC has made them optional in the implementation. I have no strong feelings either way.

The formal parameters represent the information that a client gives to the server; the return type is the information that the server returns to the client.

As an example, the following process specification says that process type `buffer` has one parameter (`max`), and two transactions, named `get` and `put`:

```
process spec buffer(int max)
    {
      trans void put(char c);
      trans char get();
    };
```

To allow the definition of functions (e.g., `c_changepriority`) that can accept processes of any type as arguments and the definition of a null process value (`c_nullpid`) that can be assigned to a variable of any process type, a predefined process type

```
process anytype
```

is provided.

3.3 Process Bodies

A process body has the form:

`process body` *process-type-name*`(`*process-parameters-names*`)`
compound statement

The process body specifies the C statements to be executed by each process of that type. Each process is a sequential program component that runs independently and in parallel with other processes. The compound statement in the process body can have automatic variables; each process of that type will get its own set of variables. Process parameters are used in the process body just as function parameters are used in function bodies. Unlike C function parameters, process parameters are not declared before the *compound statement*. This is because all the necessary type information must be supplied in the mandatory process specification. Values for the process parameters are supplied when creating a process (see discussion of the `create` operator given below).

As an example, here is a skeletal body for the process type `buffer` whose specification was given earlier:

```
process body buffer(max)
    {
      char *bufp;

      bufp = malloc(max);
      accept requests to execute get and put transactions
      free(bufp);
    }
```

A process body cannot reference global variables; this is how we discourage the use of shared memory (see Section 3). However, a process body can call functions that access, update, or return global variables; this is our concession to reality.

A function called from a process body is considered to be executing on behalf of that process. Any function can be called by processes of any type.

In a process body, the `return` statement terminates the process that executes it. This is equivalent to running off the end of the process body. Note that processes cannot return any value.

Process bodies can contain several Concurrent C extensions, such as `accept` and `select` statements. These will be described later.

3.4 Process-Valued Expressions and Process Variables

The `create` operator is used to create (instantiate) a new process of the specified type with appropriate values for the process parameters. For example,

```
process spec buffer(int max)
{
  ...
};

⋮
```

... create buffer(128) ...

⋮

creates a new process of type `buffer`, with `128` for the value of the parameter. The `create` operator returns a value of the process type being instantiated which in this case is a value of type `process buffer.` A process value identifies a specific process. A process value can be stored in a process variable of the same type:

```
process buffer b;

⋮
```

b = create buffer(128);

Concurrent C programs can have arrays of process variables, or pointers to process variables, as in

```
process buffer bufarr[10], *pb;
```

Process variables can also appear in structures.

If a process value is stored in several process variables, then all these variables refer to the same process. As with pointer values, care must be taken to avoid dangling references.

Provided that the corresponding types match, process values can be used in the following ways:

- as arguments to functions, transactions, or processes.
- as values returned by functions or transactions.
- in assignment to process variables.
- in equality or inequality tests.

The reserved name `c_nullpid` is a null (invalid) process value of type `process anytype`.. `c_nullpid` is polymorphic, in that it can be assigned to a process variable of any type. Thus `c_nullpid` is to process values as `NULL` is to pointers.

The built-in pseudo-function `c_mypid()` returns the process value of the process executing the function.

3.5 Process Creation and Activation

As mentioned before, the `create` operator creates and activates a new process. The created process becomes eligible for execution immediately. The created process is called a *child* process of the creating, or *parent*, process. Processes that have the same parent process are called *sibling* processes.

As mentioned earlier, if the `create` operator executes successfully, then it returns a process value identifying the created process; if `create` is unsuccessful, then it returns `c_nullpid.`

The `create` operator has the general form:

`create` *process-type-name*`(`*initial-values*`)` **[**`with priority(`*p*`)`**]**

where *p* is an integer expression that specifies the new process's priority relative to some standard priority. Positive values give the new process higher priority; negative values give it lower priority. If the priority is omitted, then the new process is assigned the standard priority.

Priority specification assists the implementation in the allocation of the processor when one physical processor is being shared by two or more processes. The process scheduler is free to schedule processes

subject to the following rules:

1. If two processes that are ready for execution have different priorities, then the one with the higher priority is given preference for execution.
2. If two processes that are ready for execution have the same priority, then either of them may be selected for execution.
3. No process should be indefinitely denied execution because of processes at the same or lower priority level.

Priorities should be used to improve performance, but *not* for synchronization. For example, programmers should *not* assume that a higher priority process will immediately preempt all lower priority processes.

The priority of a process can be changed by using the functions `c_changepriority` and `c_setpriority`. Function `c_setpriority` specifies an absolute priority while function `c_changepriority` specifies a relative priority. The call

```
c_changepriority(pid, p)
```

changes the priority of process *pid* by *p*, which is an integer expression; the new priority is the old priority plus *p*.

The call

```
c_setpriority(pid, p)
```

sets the priority of process *pid* to *p*.

Function `c_getpriority` is used to determine the priority of a process. The call

```
c_getpriority(pid)
```

yields the priority associated with the process identified by *pid*.

3.6 Process States and Process Termination

A process can be in one of the following three states:

active	A process becomes *active* upon creation and remains in this state while executing the statements specified in the corresponding process type body.
completed	A process becomes *completed* when it executes a `return` statement in its process body, or when it reaches the end of its body.
terminated	A process becomes *terminated* when it has completed and all the processes created by it have terminated or it executes the *ready-to-terminate* alternative defined in the section on the *select* statement.

A process can also be explicitly terminated by means of the `c_abort` function, e.g., the call

```
c_abort(p);
```

aborts process `p`. Aborting an active or completed process forces it to become terminated. That is, `c_abort` terminates all children (and grandchildren, etc.) of the indicated process. Aborting a terminated process has no effect. A process should be aborted only under extreme circumstances.

3.7 Transaction Call

A transaction call is the caller's side of a transaction. The format is similar to a function call:

process-value`.`*transaction-name*`(`*actual-parameters*`)`

Process-value is a process-valued expression. The specification for that process type must have a transaction named *transaction-name.* The types of the actual parameters must match the parameter types declared for that transaction.[4] The transaction call has the type returned by the transaction. In general, a

transaction call can be used wherever an expression of that type is allowed.

The calling process is delayed until the called process accepts the transaction (see the `accept` statement, below). The called process is given the values for transaction parameters specified by the caller. The calling process then remains suspended until the called process returns a value. This returned value becomes the value of the transaction call expression.

Note that the transaction call is directed to a *specific* process, not to an arbitrary process of that type.

Concurrent C does allow pointers as parameters in transactions. However, we discourage their use because that will require the two processes to share memory, which will be impractical in a non-shared-memory multicomputer.

3.8 Accept Statement

The `accept` statement is the receiving process's side (i.e., the server's side) of a transaction. An *accept* statement has the form

`accept` *transaction-name*`(`*formal-parameter-names*`)` **[** `suchthat` *e* **][** by *ae* **]**
[*compound statement* **]**

where *e* is a boolean expression and *ae* is an arithmetic expression; these expressions should involve the formal parameters of the *accept* statement. An `accept` statement can only appear in the body of a process. That process's specification must have a transaction declaration for *transaction-name.*

Let us first consider the `accept` statement without any accompanying `suchthat` and `by` clauses. Now if a process has one or more transaction calls outstanding for a transaction type, then an `accept` statement for that transaction type accepts one of them immediately. Transaction calls are accepted in first-in-first-out (FIFO) order. If there are no pending transaction calls, execution of the `accept` statement is delayed until a transaction call arrives.

The `suchthat` clause can be used to restrict the set of acceptable transaction requests. Only those transaction requests for which the expression *e* evaluates to true will be considered for acceptance. In the absence of a `by` clause transaction will be accepted from the restricted set in FIFO order. If *e* evaluates to false for all the outstanding transactions, the execution of the `accept` statement will be delayed until the arrival of transaction request for which *e* is true.

The FIFO order of accepting transaction requests can be altered by using a `by` clause. If a `by` clause is present, then the arithmetic expression *ae* is evaluated for each transaction request that can be accepted,, and the request with the minimum value is accepted first [ANDR81, ANDR82].

Once a transaction call has been accepted, the process executes the compound statement that forms the body of the `accept` statement. Within that compound statement, the formal parameter names become variables that hold the parameter values given by the caller. (The types of these variables are given in the transaction declaration in the process specification.) These parameters are passed by value.

The calling process is delayed until the `accept` statement terminates by executing a `treturn` statement:

`treturn` *expression*`;`

The value of the expression is returned to the calling process. The process containing the `accept` statement goes on to execute the next statement (after the body of the `accept`) while the process issuing the transaction call becomes free to resume execution.

If the transaction is declared as returning a value of type `void`, the `accept` statement terminates by using a `treturn` statement with no value, or by just running off the end of the compound statement.

---

4. Rules used for matching of transaction parameters are the same as those used by C for matching function parameters.

Note that the scope of a parameter variable is limited to the body of the `accept` statement. To use a parameter outside of the `accept` statement, assign it to some variable with higher scope.

An `accept` statement can only be used in a process body. This ensures that only processes of the right type will execute an `accept` statement for a transaction. The `accept` statement cannot be used in a function, even if that function is only called by processes of one type. One reason for this restriction is that, in general, the compiler does not know which processes can call an arbitrary function.

3.9 Delay Statement

A process can delay itself by executing a statement of the form

```
delay duration;
```

where *duration* is a floating-point expression that specifies the amount of the delay in seconds. The actual delay may be more, but not less, than the requested delay.

3.10 Select Statement

The `select` statement allows a process to non-deterministically wait for the first of several events. The syntax is:

```
select {
   [(guard1):] alternative1
or
   [(guard2):] alternative2
or
.
.
.
or
   [(guardn):] alternativen
[otherwise statements]
}
```

A *guard* is a boolean expression (non-zero is true and 0 is false). The order in which guards will be evaluated is unspecified. Thus we suggest that guards be kept as simple as possible (e.g., guards should not make transaction calls!), and side-effects in guards should be avoided.

A *select statement alternative* can be one of the following, depending on the first statement in the alternative:

1. an `accept` statement, optionally followed by other statements (an *accept* alternative),
2. a `delay` statement, optionally followed by other statements (a *delay* alternative),
3. the keyword `terminate` followed by a semicolon (a *ready-to-terminate* alternative), or
4. the keyword `immediate` followed by a list of statements (an *immediate* alternative)

The `accept` and `delay` statements were described earlier.

The `select` statement executes one and only one of the alternatives. Once chosen, all statements in that alternative are executed, and flow resumes after the `select` statement. The following rules determine which alternative will be taken:

1. If there is an *accept* alternative with a true guard (non-zero or omitted), and if a corresponding transaction call that satisfies the `suchthat` expression, if given, is waiting, take that *accept* alternative (preference is given to process-interaction statements).
2. Otherwise, if there is an *immediate* alternative with a true guard, take that alternative.
3. Otherwise, if there is a *ready-to-terminate* alternative with a true guard, and if the termination criteria are satisfied (see below), then take the *terminate* alternative. This terminates the process.

4. Otherwise, if there is an `otherwise` clause, take it.
5. Otherwise, let *x* be the lowest delay specified by a *delay* alternative with a true guard, or infinity if there are no such *delay* alternatives. Then if one of the following events occurs within the next *x* seconds, then take the corresponding alternative:
    a. The arrival of a transaction call for one of the transactions for which there is an *accept* alternative with a true guard.
    b. The occurrence of the *termination conditions* (discussed below), if there is a *ready-to-terminate* alternative with a true guard.

    If none of these events occur within *x* seconds, take the *delay* alternative.

It is considered to be an error if none of the guards are true, and if there is no `otherwise` clause.

The *termination conditions* mentioned above are true if all of the following conditions are true:

- All the children of this process have terminated.
- All sibling processes (other processes created by the parent process) have terminated, or are waiting at a *ready-to-terminate* alternative and their children have terminated.
- The parent process has completed execution.[5]
- This process has no pending transaction calls.

Let us conclude the discussion of the `select` statement with a few notes:

- Concurrent C implementations are free to select any strategy for implementing the `select` statement, provided the semantics are equivalent to those described above. For example, an implementation might not evaluate all the guards. Consequently, we suggest that programmers avoid side-effects in guards.
- If there is an `otherwise` part, then a `delay` alternative in that `select` statement is useless; the `otherwise` part will always be executed in preference to the `delay` alternative.
- The `accept` statement for an *accept* alternative may have `suchthat` and `by` clauses. The `accept` alternative will be chosen only if the `suchthat` expression, if any, is satisfied. Once that *accept* alternative has been chosen, the `by` clause determines which of the pending calls will be accepted. The `by` clause does *not* determine whether the alternative is taken.

3.11 Timed Transaction Call

The timed transaction call allows the process requesting the transaction to withdraw the transaction call if the process named does not accept the transaction within the specified period.

The timed transaction call is an expression of the form

`within` *duration* `?` *p*`.`*t*`(`*actual-parameters*`)` `:` *expression*

where *duration* is floating point expression, *p* is a process-valued expression and *t* is a transaction name. If it is possible to execute the specified transaction before the delay *duration*, then the value of the timed transaction call is the value returned by the transaction; otherwise, its value is *expression*.

3.12 Transaction Pointers

A transaction pointer refers to a specific transaction associated with a specific process. Transaction pointers are similar to function pointers.

---

5. Without this condition, a process may terminate before its parent has had a chance to interact with it or before the siblings created by the parent have had a chance to interact with it.

Transaction pointer declarations specify the result type and parameter types associated with the transaction, but are independent of the process type. Not including the process type allows a pointer to refer to identical transactions belonging to different process types. Thus a transaction pointer could refer to transactions of different process types, as long as those transactions had the same parameter types and return-value type.

Transaction pointers declarations are similar to function pointer declarations except that these declarations must be prefixed by the keyword `trans`:[6]

`trans` *type-specifier* *transaction−pointer−declarator*$_1$`,` `...,` *transaction−pointer−declarator*$_1$`;`

where *transaction-pointer-declarator* is identical to declarator for a function pointer [RITC80].

As an example of a transaction pointer, consider[7] a process of type `user` that expects replies to its queries to a database process to be sent to its transaction `answer`. It could pass a pointer to transaction `answer` as an argument to the database process (as part of the query). The database process will use this transaction to return the answer.

Here is a skeleton for the `user` process:

```
process spec user(database d)
{
  ...
  trans message answer(...)
  ...
};

process body user(d)
{
  ...
  d.query(..., mypid().answer, ...);
  ...
  accept answer(...) { ... }
  ...
}
```

Here is a skeleton for the `database` process:

6. The syntax of a pointer to a transaction that returns a transaction looks ugly:

7. FEEL FREE to change the EXAMPLE as you see fit. This is just a first crack.

```
process spec database(...)
{
  ...
  trans void query(..., trans message (*answer)(), ...);
  ...
};

process body database()
{
  trans message (*answer)();
  ...
  accept(..., answer, ...)
  {
    ...
    reply = answer;
    ...
  }
  ...
  reply(...);
  ...
}
```

3.13 Process Scheduling

The points at which the scheduler is invoked (for processes on the same machine) is left to the implementation. For example, an implementation may use a round-robin scheduling strategy modified to take into account process priorities.

When a new process is to be scheduled, the process with the highest priority is always the first one to be scheduled. If there is more than one process with the same high priority, then one of these processes is arbitrarily selected.

3.14 Interrupts and Transactions

Interrupts can be associated with transactions so that the occurrence of an interrupt results in a call to the transaction associated with the interrupt. The call issued can be an unconditional transaction or a timed transaction call depending upon the nature of the interrupt.

An interrupt is associated with a transaction by using the library function `c_associate`, an implementation-dependent function. One of the arguments to this function will be a pointer to the transaction with which the interrupt is to be associated. Details of `c_associate` should be provided by the Concurrent C implementation.

3.15 Number of Outstanding Transaction Requests

Pseudo-function `c_transcount` returns the number of outstanding requests for a specific transaction, i.e., the call

`c_transcount(`*pid*`, `*t*`)`

returns the number of outsanding requests associated with transaction *t* of process *pid*; Note that value returned by function `c_transcount` includes pending timed transaction calls, which can be withdrawn by the requesting process. Consequently, the value returned by `c_transcount` might not be accurate because of withdrawn requests.

3.16 Process States

`c_active(`*p*`)`	returns 1 if process *p* is active; otherwise it returns 0.
`c_completed(`*p*`)`	returns 1 if process *p* has completed; otherwise it returns 0.

`c_valid(`*p*`)` returns 1 if *p* refers to an active or completed process; otherwise it returns 0 (the process has terminated or *p* is an invalid value).

3.17 Building Library and Server Processes

The process accepting transaction requests does not have to know the name of the process issuing the transaction call. This asymmetry allows the building of library and server processes.

3.18 Reserved Keywords

Here is a list of the new keywords added for Concurrent C:

```
accept      body        by        create
delay       immediate   or        otherwise
priority    process     select    spec
suchthat    terminate   trans     treturn
with        within
```

4. Examples

To give the reader a flavor of Concurrent C programs,[8] we will give two examples: the *dining philosophers* problem and a database lock manager.

We selected the dining philosophers problem because it has been studied extensively in the computer science literature. It is used as a benchmark to check the appropriateness of concurrent programming facilities and of proof techniques for concurrent programs. It is interesting because, despite its apparent simplicity, it illustrates many of the problems, such as shared resources and deadlock, encountered in concurrent programming. The forks are the resources shared by the philosophers who represent the concurrent processes.

We selected the lock manager because it is representative of a real concurrent programming problem. It also illustrate the power of the `suchthat` clause in Concurrent C.

4.1 The Mortal Dining Philosophers [GEHA84b]

Five philosophers spend their lives eating spaghetti and thinking. They eat at a circular table in a dining room. The table has five chairs around it and chair number `i` has been assigned to philosopher number `i` ($0 \le i \le 4$). Five forks have also been laid out on the table so that there is precisely one fork between every adjacent two chairs. Consequently there is one fork to the left of each chair and one to its right. Fork number `i` is to the left of chair number `i`.

Before eating, a philosopher must enter the dining room and sit in the chair assigned to her. A philosopher must have two forks to eat (the forks placed to the left and right of every chair). If the philosopher cannot get two forks immediately, then she must wait until she can get them. The forks are picked up one at a time. When a philosopher is finished eating (after a finite amount of time), she puts the forks down and leaves the room.

The five philosophers and the five forks will be implemented as processes. On activation, each philosopher is given an identification number (0-4) and the process values of the forks she is supposed to use. Each philosopher is mortal and passes on to the next world soon after having eaten 100,000 times (about three times a day for 90 years).

The specification of the `philosopher` and `fork` processes is

---

8. CHANGED/NEW Check Wording

```
process spec fork()
{
  trans void pick_up();
  trans void put_down();
};

process spec philosopher(int id, process fork left, process fork right);
```

The bodies of the `philosopher` and `fork` processes are

```
#define LIFE_LIMIT 100000
#include "dining.h"  /*contains the process specifications*/

process body philosopher(id, left, right)
{
  int times_eaten;

  for (times_eaten = 0; times_eaten != LIFE_LIMIT; times_eaten++) {
    /*think for a while; then enter dining room and sit down*/

    /*pick up forks*/
      right.pick_up();
      printf("PHIL %d: PICK UP FORK %d\n",id, id+1%5);
      left.pick_up();
      printf("PHIL %d: PICK UP FORK %d\n",id, id);
    /*eat*/
      printf("Philosopher %d: That was delicious -- Thank You\n", id);
    /*burp*/

    /*put down forks*/
      left.put_down();
      printf("PHIL %d: PUT DOWN FORK %d\n",id, id);
      right.put_down();
      printf("PHIL %d: PUT DOWN FORK %d\n",id, id+1%5);

    /*get up and leave dining room*/
  }
  printf("Philosopher %d: SEE YOU IN THE NEXT WORLD\n", id);
}

process body fork()
{
  for (;;)
    select {
      accept pick_up();
      accept put_down();
    or
      terminate;
    }
}

main()
{
  process fork f[5];
  process philosopher p[5];
```

```
  int j;    /*loop variable*/

  /*create the forks and philosophers; the forks must be*/
  /*created before the philosophers*/

    for (j=0; j<=4; j++)
      f[j] = create fork();

    for (j=0; j<=4; j++)
      p[j] = create philosopher(j, f[j], f[(j+1)%5]);
}
```

Once the philosophers have terminated, the forks have nothing else to do; because each fork is waiting at a *select* statement with a *terminate* alternative, and the parent process (i.e., `main`) has completed, they all terminate. This allows the `main` process to terminate which completes execution of the Concurrent C program.

Note that the output of the philosophers can get mixed up if execution of a philosopher is interrupted when it is trying to write to standard output and another philosopher then writes to the standard output.

4.2 Lock Manager

5. Concurrent C Design Decisions

In this section[9] we will discuss the rationale for the design decisions we made for Concurrent C.

5.1 Concurrent Programming Model

The most important decision was the selection of the concurrent programming model on which to base Concurrent C. We haved discussed the rationale for this earlier in Section 2. To summarize, we selected a message passing model so that we could implement Concurrent C efficiently on a multicomputer.

5.2 Limitations of UNIX™ Concurrency

Concurrent programs can be (and have been!) written in C, using multiple UNIX processes[10] and UNIX operating system calls. However, this approach has several disadvantages:

1. Only limited interaction is possible between UNIX processes. For example, the primary UNIX inter-process communication mechanism is the pipe. Pipes are elegant, and are fine if the process interaction can be reduced to a simple linear stream (e.g., `troff` preprocessors). However, pipes are not a general inter-process communication mechanism. In particular, they are not appropriate for replies from a server process. There is too much overhead associated with setting up a pipe for just one reply, e.g., the server must open a pipe, write the reply and then close the pipe.
2. While the UNIX shell provides an elegant interface for using pipes, the underlying system calls–`fork, exec, pipe`–are not easy to use. If not used with care, they can lead to errors.
3. A UNIX process cannot simultaneously wait for multiple events.
4. Some versions of the UNIX system provide message passing facilities [UNIX82]. All messages are byte strings that are interpreted by the receiving process. However, such a scheme does not allow compile-time checking of message types. Furthermore, the message passing facilities are not standardized and they are based on the asynchronous message passing model about which we have

---

9. CHANGED/NEW

™ UNIX is a trademark of AT&T Bell Laboratories.

10. The terms *process* and *UNIX process* will be used to differentiate between Concurrent C processes and processes on the UNIX operating system, respectively.

reservations (see next section).

5. A fast process context switching mechanism is important. This will encourage programmers to write small, simple, modular processes–just as a fast function call mechanism encourages programmers to write small, simple, modular functions. The UNIX process context switch time–about 1 ms on a Vax 11/780–is too high. This high cost results from the cost of changing address spaces: purging translation buffers, invalidating the cache, swapping processes, and so on.

6. Many versions of UNIX enforce a limit on the number of processes that one user can run at a time. Typically this limit is about 20 processes, which is too small for many concurrent programming applications.

5.3 Concurrent Programming Facilities via Language Extensions versus Library Functions

1. It would be clumsy and inelegant to provide some of the facilities, e.g., `select` statement.

2. The concurrent parts of a program will not be easily identifiable>

3. The semantics of these subprograms and the interaction of concurrency with the facilities in the rest of the programming language may not be clearly or precisely defined.

4. The subprograms providing concurrency are often local and ad hoc extensions to the sequential programming language, resulting in programs that are difficult, if not impossible, to port.

5. A compiler may not be easily able to optimize concurrent programs well, since it will be tuned to optimizing sequential programs [PRA83].

5.4 Transaction Pointers

Transacition pointers and function pointers are not interchangable because

1. it would change semantics of C (which we wanted to avoid)

2. complicate implementation

3. negative impact on portability.

5.5 `suchthat` clause

1. Efficiency

5.6 No Types in `accept` Statements

6. Differences Between Concurrent C and Ada

The concurrent programming facilities in Ada and Concurrent C are both based on the rendezvous concept. However, the implementation of the rendezvous concept in Concurrent C is quite different from that in Ada. Although many of the differences are minor (e.g., syntax changes), some of the differences are major and can have a significant impact on the programming style and implementation efficiency:

Transaction Calls:
In Concurrent C, transaction calls are similar to function calls. In Ada, they are similar to procedure calls. Thus in Concurrent C, transaction calls can be used in arbitrary expressions, while in Ada, they can only occur as statements.

Ordering of Transaction Calls:
In Concurrent C, the programmer can specify the order in which transaction calls are to be accepted (via the `by` clause). In Ada, calls must be accepted in FIFO order. This difference between Concurrent C and Ada will have a significant influence on how schedulers are designed in the two languages. For example, in Ada, a process that must run transactions in non-FIFO order will require two rendezvous for each service request [GEHA84b, GEHA84c]. In Concurrent C, such a process only needs one rendezvous per service request.

In Concurrent C, the `by` clause does incur some extra overhead. However, this is not significant if the number of pending transaction calls is small. Furthermore, accomplishing the same thing in Ada

requires an extra rendezvous, which is more expensive than the extra overhead of the Concurrent C `by` clause.

Selective Acceptance of Transaction Calls:
The `suchthat` clause can be used to restrict the set of acceptable transaction requests. Only transaction requests with parameters that satisfy the `suchthat` expression are considered for acceptance. There is no comparable mechanism in Ada.

The Select Statement:
Concurrent C allows *immediate* alternatives in `select` statements; Ada does not. This restriction is one of the reasons why Ada has a polling bias [GEHA84b].

Transaction Pointers:
There is no mechanism in Ada for specifying a pointer to a transaction.[11] Consequently, it is not possible for a process to dynamically specify a process interaction point to another process. Moreover, it is not possible for a specific transaction call to refer to an identical transaction in processes with different types.

Arrays of Transactions:
Ada provides arrays of entries; Concurrent C does not provide arrays of transactions. In Ada, entry arrays are generally used to implement a non-FIFO scheduling policy. The `by` and `suchthat` clauses make array entries unnecessary in Concurrent C. Furthermore, in Ada, array entries require polling, which is wasteful of system resources [GEHA84b].

Process Parameterization:
Concurrent C processes can be parameterized, allowing the initialization of processes at creation time. Process initialization at creation time is not possible in Ada; consequently, initialization parameters must be supplied explicitly to the process after it has been activated. The overhead involved includes declaring an entry that is used for initialization, and one rendezvous for supplying the initial values.

Process Activation:
Process activation is implicit in Ada, while in Concurrent C it is explicit. Thus in Concurrent C, the programmer has more control over when processes are instantiated; however, the programmer must explicitly instantiate the processes (by using the `create` operator).

Process Ownership:
In Concurrent C, processes are owned by the process which created them; in Ada they are owned by the program unit, i.e., a block, subprogram or process, that created them.

Global Data:
To discourage the use of global variables (i.e., shared memory), Concurrent C processes cannot directly refer to global variables. Ada has no such restriction. Of course, Concurrent C processes can reference global variables by calling functions.

Process Nesting:
Ada allows nested processes; Concurrent C processes cannot be nested. This is in the spirit of C, which does not allow nested functions.

Process Declaration:
In Concurrent C, process types must be declared in order to declare processes. In Ada, processes can be declared directly, in addition to using process types.

Priority Specification:
In Concurrent C, priorities are specified when creating a process; Different processes of the same

11. NEW/CHANGE

process type can have different priorities. Moreover, process priorities can be changed dynamically.

In Ada, all processes instantiated from a process type must have the same priority, because the priority is associated with the process type. Moreover, specification of process priority in Ada is only a suggestion to the compiler; Ada compilers are not required to implement process priorities.

Interrupts and Transaction Calls:

In Ada, declarations are used to associate interrupts with transaction calls; on the other hand, Concurrent C uses a library function to make this association. The advantage of using a library function is that associations can be made dynamically. Moreover, this allows associations to be changed or discontinued; this is not possible in Ada.

7. Problem with C Library Functions that Update Shared Memory

We would like Concurrent C programs to be able to call standard UNIX library functions–things like `malloc` and the `stdio` library. However, some of these functions update shared memory, and will require modifications before they will work in a concurrent environment. For example, suppose that the scheduler decides to switch processes in the middle of a `malloc` call, just after `malloc` has partially updated its storage allocation data. If the next process also calls `malloc`, the result will be chaos.[12]

There are several solutions. One is to provide a library with concurrent versions of functions like `malloc`. For example, the concurrent `malloc` could issue a transaction call to a storage allocator process, which is the only process that updates the storage allocation data. Unfortunately, this solution requires us to identify those library functions that use shared data, and to maintain separate versions of those functions. Furthermore, the concurrent versions of those functions will be less efficient than the simple sequential versions.

Another solution is to make all calls to sequential C library functions that update shared memory to be *atomic* so that they are never interrupted in the middle.[13]

8. Conclusions

The concurrency model in Concurrent C is based on the rendezvous concept. Many of the differences between the Concurrent C and Ada implementations of the rendezvous are a result of trying to eliminate the polling bias in Ada [GEHA84b], to increase programmer control of scheduling, and to provide the programmer with more control of the concurrency (process parameterization and explicit process activation).

Concurrent C can be used for a variety of applications:

1. To implement parallel algorithms, such as simulation programs and protocols.
2. To write genuinely distributed applications, such as distributed databases.
3. To write real-time programs.
4. To write dedicated programs, including programs such as device drivers, that execute on a ‘‘bare’’ machine.

---

12. In theory, the same problems can occur in ordinary sequential C programs that use signals. E.g., suppose a signal interrupts a `malloc` call, and somehow `malloc` is called again before the first call is resumed. In practice, typical signal handlers are very simple, so this does not happen.

13. Our current implementation of Concurrent C makes all sequential C library functions atomic by ensuring that the Concurrent C scheduler never dispatches away from a process when it is in such a function. Every second, the scheduler gets an interrupt (a UNIX alarm signal), and normally suspends the currently running Concurrent C process and starts another. However, if the program counter at the time of interrupt was in a UNIX library function, the scheduler does not force a process switch. You can decide for yourself whether this is solution is simple and elegant, or the worst piece of sleaze since programmers learned how to abuse Fortran EQUIVALENCE statements.

5. To implement operating systems.

The concurrent programming facilities in Concurrent C are subject to revision. The design of these facilities has been influenced by the fact that the first implementation of Concurrent C will use a preprocessor and we wanted to simplify the preprocessing. We plan to remove any awkwardness resulting from this decision prior to implementing a full-fledged compiler.

We hope that writing real programs in Concurrent C will lead to an evaluation of these facilities followed by modifications of these facilities (if necessary). Based on our limited experience with Concurrent C and discussions about its facilities, we already have several possible candidates for changing and extending Concurrent C:

1. Allow an `accept` statment in a function that can be only be called from a process in which such an `accept` statement would be legal.
2. The transaction pointer mechanism would allow transactions to be associated with interrupts by calling some run-time support function. This would have several advantages over the Ada technique, which statically associates interrupt addresses with a process type:
    1. Association of transactions with interrupts would be done dynamically.
    2. The association could be done within or without the process body.
    3. Different instantiations of a process could be associated with different interrupts.
    4. A variety of interrupt mechanisms could be handled in Concurrent C because the function that does the association can be implemented differently on different computer architectures.
    5. Multiple interrupts could be mapped to the same transaction.
3. Processes with the *ready-to-terminate* alternative terminate when all siblings (that are alive) are ready to terminate and the parent process has completed. We are not happy with the sibling part of this condition because it forces a process to wait for siblings who may never interact with it.
4. Arrays cannot be passed by value in C; instead a pointer to the first element is passed. This is not be acceptable for a distributed implementation of Concurrent C because it implies the need of shared memory for an efficient implementation. (Of course, arrays can be packed in structures and then passed as arguments; we do not like this approach because it tries to skirt around the problem instead of solving it; moreover, it does not allow passing of variable-sized arrays.)

    We would like to provide a mechanism by which arrays are passed by value.
5. Process initialization at creation time[14]

We have implemented Concurrent C on the UNIX system running on a single processor.

We are now planning to implement a distributed version of Concurrent C that will allow a Concurrent C program to run on multiple processors. This will give us a vehicle to explore issues involved in a multicomputer implementation. For example:

1. How should the multicomputer computer architecture be conveyed to the Concurrent C compiler?
2. Should the programmer specify on which computers the different processes are to be executed, or should the compiler decide on its own?
3. Should groups of processes be allowed to share global data in a controlled manner, or should processes be prohibited for sharing data?

---

14. NEW/CHANGE

Acknowledgement

B. W. Ballard, T. A. Cargill, R. F. Cmelik, M. D. Durand, A. R. Feuer, D. Gay, D. D. Hill, B. W. Kernighan, M. P. Leland, D. E. Perry, R. Mascitti, J. Schwartz, B. Smith-Thomas, D. Swartwout, and some anonymous referees gave us detailed comments. We are grateful for their suggestions and criticisms.

References

Some of these references are not cited in the text.

ANDR81 Andrews, G. R. Synchronizing Resources. *TOPLAS*, v3, no. 4, pp. 405-430, October 1981.

ANDR82 Andrews, G. R. The Distributed Programming Language SR—Mechanisms, Design and Implementation. *Software—Practice and Experience*, v12, pp. 719-753, 1982.

BOUR82 Bourne, S. R. *The UNIX System*. Addison-Wesley Publishing Co., 1982.

DOD79 *Rationale for the Design of the Ada Programming Language*. *SIGPLAN Notices*, v14, no. 6, part B, June 1979.

DOD83 *Reference Manual for the Ada Programming Language*. United States Department of Defense, January 1983.

GEHA84a Gehani, N. H. and W. D. Roome. Concurrent C—Programming Examples. Internal Memorandum, AT&T Bell Labs, 1984.

GEHA84b Gehani, Narain. *Ada: Concurrent Programming*. Prentice-Hall, 1984.

GEHA84c Gehani, N. H. and T. A. Cargill. Concurrent Programming in the Ada Language: The Polling Bias. To be published in *Software—Practice and Experience*.

GEHA84d Gehani, N. *C: An Advanced Introduction*. Computer Science Press, 1984.

GENT81 Gentleman, W. M. Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept. *Software—Practice and Experience*, v11, pp. 435-466, 1981.

HOAR78 Hoare, C. A. R. Communicating Sequential Processes. *CACM*, v21, no. 8, pp. 666-677, August 1978.

JOHN83 Johnson, S. C. and B. W. Kernighan. The C Language and Models for Systems Programming. *Byte*, v8, no. 8, pp. 48-60, August 1983.

KERN78 Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

LISK82 Liskov, B. and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programming. *Proceedings of the Ninth Symposium on the Principles of Programming Languages*, pp. 7-19, January 1982.

PRAT83 Pratt, V. Five Paradigm Shifts in Programming Language Design and Their Realization in Viron, a Dataflow Programming Environment. *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983.

RITC80 Ritchie, D. M. The C Programming Language—Reference Manual. AT&T Bell Labs, September 1980.

STRO82 Stroustrup, B. A Set of C Classes for Coroutine-Style Programming. Computing Science Technical Report No. 90, AT&T Bell Laboratories, July 1982.

STRO83 Stroustrup, B. Adding Classes to the C Language: An Exercise in Language Evolution. *Software-Practice and Experience*, v13, pp. 139-161, 1983.

STRO84 Stroustrup, B. The C++ Reference Manual. Computing Science Technical Report No. 108, AT&T Bell Laboratories, January 1984.

UNIX82 *UNIX System User's Manual (Release 5.0).* AT&T Bell Laboratories, June 1982.

UNIX83 *UNIX Programmer's Manual (4.2 BSD).* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, August, 1983.

Implementation Restrictions

1. Transactions cannot return structures because C functions that return structures are not reentrant.

APPENDIX: Changes to Concurrent C

Changes up to Jan 31, 1985

1. Type `process anytype` added.
2. Function names and specifications changed slightly for uniformity (e.g., prefix `c_` added) and new functions added (e.g., `c_getpriority`, `c_changepriority` and `c_setpriority`)
3. Keyword `immediate` added (replaces the use of a *null* statment to specify an *immediate* alternative.
4. Deleted comments about initial startup because they are just implementation notes.
5. `suchthat` clause has been added to the `accept` statement.
6. Keyword `suchthat` added.

Changes up to Mar 31, 1985

1. Parameter names must now be supplied in process specifications for both the process and transaction formal parameters.
2. Transaction pointers have now been added.

CONTENTS

Concurrent C*

N. H. Gehani
W. D. Roome

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Concurrent programming is becoming increasingly important because multicomputers, particularly networks of microprocessors, are rapidly becoming attractive alternatives to traditional maxicomputers. Effective utilization of such network computers requires that programs be written with components that can be executed in parallel.

The C programming language [KERN78] does not have concurrent programming facilities. Our objective is to enhance C so that it can be used to write concurrent programs that can run efficiently on both single computers and multicomputers. Our concurrent extensions to C are based on the *rendezvous* concept. These extensions include mechanisms for the declaration and creation of processes, for process synchronization and interaction, and for process termination and abortion. We give a rationale for our decisions and compare Concurrent C extensions with the concurrent programming facilities in Ada.

Concurrent C has been implemented on the UNIX system running on a single processor.

---

* Revised on January 18, 1985; again on January 31, 1985—clerical changes + addition of the `suchthat` clause.