Concurrent C: real-time programming and fault tolerance*

by N.H. Gehani

Concurrent C is an upward-compatible parallel extension of C which runs on a variety of uniprocessors and multiprocessors. A Concurrent C program consists of a set of processes which execute in parallel and interact with each other by sending messages. Fault-Tolerant (FT) Concurrent C, an extension of Concurrent C, is a tool for writing fault-tolerant distributed programs, based on the replication of critical processes. All details of interaction with replicated (fault-tolerant) processes are handled by the FT Concurrent C runtime system. Consequently, writing fault-tolerant distributed programs is almost the same as writing ordinary distributed programs. In this paper, we briefly describe Concurrent C, discuss its real-time facilities and describe FT Concurrent C.

1 Introduction

Concurrent C [2] is a superset of C which provides parallel programming facilities. It has also been integrated with C++, which extends C to provide data-abstraction facilities [2, 3]. A concurrent C program consists of a set of components called *processes* which execute in parallel. Processes interact by means of transactions† that can be synchronous or asynchronous. Synchronous transactions implement the extended rendezvous concept (as in Ada); two processes interact by first synchronising, then exchanging information (bidirectional information transfer) and, finally, by continuing their individual activities. A process calling a synchronous transaction is forced to wait (unless it times out) until the called process accepts the transaction and performs the requested service. With asynchronous transactions, the caller does not wait for the called process to accept the transaction; instead, the caller continues with other activities after issuing the transaction call. Information transfer in asynchronous transactions is unidirectional; from the calling process to the called process.

Multiprocessor implementations of Concurrent C led us to address the issue of what happens to a Concurrent C program running on a multiprocessor in the presence of partial hardware failure. The result was the design and implementation of Fault-Tolerant (FT) Concurrent C [4], an extension of Concurrent C. FT Concurrent C is a tool for writing fault-tolerant distributed programs that can run with full functionality in the presence of partial hardware failure. FT Concurrent C provides fault tolerance by allowing the programmer to replicate critical processes. All details of interaction with replicated (fault-tolerant) processes are handled by the FT Concurrent C runtime system. As far as the user is concerned, interacting with a replicated process is the same as interacting with an ordinary process, and writing fault-tolerant distributed programs is almost the same as writing ordinary distributed programs.

In this paper, we describe Concurrent C, discuss the real-time facilities provided by it and illustrate their use with examples, describe FT Concurrent C and give examples to illustrate its use. We will assume that the reader is familiar with the C programming language.

2 Concurrent C: summary and an example

Concurrent C extends C for parallel programming by providing facilities for

- defining processes; a process definition consists of a process specification, i.e. a process type, and a process body;
- creating processes, using the **create** operation;
- specifying the processor on which a process is to run, using the **processor** clause of the **create** operator;
- specifying, querying and changing process priorities,

* This paper is a modified and expanded version of a previously presented paper [1].

† To avoid confusion with 'database transactions', please note that we will use the term 'transaction' to mean a Concurrent C process interaction. Transactions are like remote procedure calls with one important difference; the receiving process can schedule acceptance of the calls.

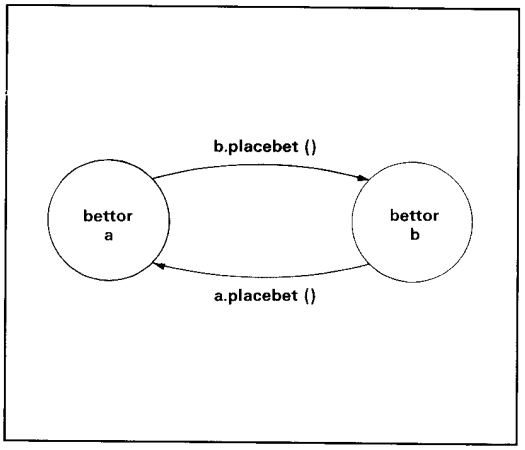

Fig. 1. The bettors

using the **priority** clause of the **create** operator and library functions;

- synchronous transactions, for synchronous bidirectional information transfer;
- asynchronous transactions, for asynchronous unidirectional information transfer;
- delays and timeouts, using the **delay** statement and the **within** operator;
- interrupt handling, using the **c_associate** function which associates interrupts with transaction calls;
- waiting for a set of events, using the **select** statement which allows a process to wait for a set of events, such as the arrival of transactions and the expiration of a delay, without the necessity of polling;
- accepting transactions (using the **accept** statement) in a user-specified order (with the **by** clause of the **accept** statement) and selectively, using guards of the **select** statement and the **suchthat** clause of the **accept** statement;
- process abortion, using the **c_abort** function;
- collective termination automatically; when all processes in a program are waiting at a **terminate** alternative of the **select** statement, then the whole program terminates automatically).

We will now show you a Concurrent C program that models a very simple two-person betting game [2]. Each person is modelled as a process of type **bettor**. The two processes take turns betting. To place a bet, one process calls the other with its bet. A player's bet is his opponent's last bet plus a random number between 1 and 100, provided the total does not exceed the player's betting limit. If it does, the player gives 0 as a bet, signifying that he has lost. One process is designated as the first bettor and initially calls the other with a bet of 1. After this, the processes alternate between waiting for the next bet (waiting for a transaction call) and placing a bet (making a transaction call). The processes continue until one of them places a bet of 0. The first bettor prints the results.

Below is the specification of the **bettor** process.

File: bettor.h

```
process spec bettor(int first, int limit)
{
  trans void playwith(process bettor player);
  trans void placebet(int bet);
};
```

bettor processes are created with two parameters; the first specifies whether or not the process represents the first player, and the second specifies the player's betting limit. Each bettor process has two synchronous transactions: **playwith** and **placebet**. The first transaction informs the **better** process of its opponent, and the second transaction **placebet** is used by the opponent to place a bet (Fig. 1).

This is a body of the **bettor** process.

File: bettor.cc

```
# include "bettor.h."
process body bettor(first, limit)
{
  int mybet = 1, hisbet = 1;
  process bettor opponent;
  accept playwith(player)
    opponent = player;
  if (first)
    opponent.placebet(mybet);
  while (mybet > 0 && hisbet > 1)  {
    accept placebet(bet)
      hisbet = bet;
    if (hisbet > 0)  {
      mybet = hisbet + 1 + rand()%100;
      if (mybet > limit)
        mybet = 0;
      opponent.placebet(mybet);
    }
  }
  if (first)
    if (mybet > 0)
      printf("I won; last bet %d\n",mybet);
    else
      printf("I lost; last bet %d\n",hisbet);
}
```

Each **bettor** process first waits to get the id of its opponent **bettor** process and then it alternates between placing a bet and accepting a bet; the first **bettor** process starts by placing a bet while the second **bettor** process starts by accepting a bet.

Finally, this is the main process.

File: bet-main.cc

```
# include "bettor.h"
# define BETTING_LMT 1000
# define FIRST_PLAYER 0
main()
{
  process bettor a, b;
  srand(time(0)); /* set seed for rand() */
  a=create bettor (FIRST_PLAYER==0, BETTING_LMT);
  b=create bettor (FIRST_PLAYER==1,BETTING_LMT);
  a.playwith(b); b.playwith(a);
}
```

The **main** process creates the two **bettor** processes and records their ids (the values returned by the **create** operator). It then calls transaction **playwith** of each **bettor** process to inform it of its opponent.

Although we have shown you a very simple example of a game playing program, this program structure can be used for implementing more sophisticated game playing programs.

3 Real-time facilities

Concurrent C provides several facilities for handling events (process interactions) in a timely manner and for handling interrupts.

3.1 *Accepting transaction calls*

Concurrent C allows transactions to be accepted in a user-specified order; thus, the user can accept urgent transaction calls first. According to Bloom [5], a process interaction mechanism is fully expressive only if process interaction requests (transaction calls in Concurrent C) are accepted based on the following information:

- ☐ name of the transaction.
- ☐ the order in which transaction calls are received.
- ☐ arguments of the transaction.
- ☐ state of the called process.

Concurrent C satisfies these criteria. Transaction requests are accepted with the **accept** statement which specifies the transaction name. By default, transaction calls are accepted in FIFO (First-In First-Out) order. The **by** clause of the **accept** statement allows the acceptance order to be changed. The **suchthat** clause of the **accept** statement allows selection of the transaction call to be based on the arguments of the transaction. Finally, the **suchthat** clause and the guards of the **select** statement allow the selection of the operation to be based on the state of the called process.

3.2 *Asynchronous message passing*

Originally, Concurrent C had only synchronous message passing facilities. Eventually, after much discussion, asynchronous message passing facilities were added. Synchronous and asynchronous message passing facilities are equivalent, in that one can be implemented in terms of the other. Consequently, most parallel programming languages do not support both kinds of message passing.

In practice, both synchronous and asynchronous message passing facilities are desirable [6, 7]. For example, SR [8] also has both kinds of message passing facilities. Synchronous message passing, as provided by Concurrent C transactions, provides a higher level of abstraction than that provided by asynchronous message passing, and it is efficient especially for bidirectional information transfer. Synchronous message passing is also easier to implement and less prone to programming errors. On the other hand, asynchronous message passing has the following advantages.

- **Programming ease:** in the absence of asynchronous message passing facilities, asynchronous message passing can be simulated by introducing buffer processes. However, these extra processes increase program complexity.
- **Flexibility:** asynchronous message passing gives the maximum flexibility to the programmer, which can be used to maximise concurrency and speed up responses to critical events.
- **Pipelining and throughput:** asynchronous message passing cuts down on the time required for sending messages because message transmission can be 'pipelined', i.e. overlapped with the time to compute the message. In addition, a smart implementation can pack multiple asynchronous transaction calls, destined for the same processor, in a single buffer and send it as one inter-processor message [6]. Asynchronous transaction calls also allow one process to have multiple transaction calls outstanding at another process such as a disk driver; the disk driver can then accept these calls in an optimal order. For example, if a process makes the following asynchronous write requests to a disk scheduler process

```
disk.request(blk_no1, WRITE, buf1, ...);
disk.request(blk_no2, WRITE, buf2, ...);
disk.request(blk_no3, WRITE, buf3, ...);
```

then these requests will be queued and will all be available to the disk scheduler. The disk scheduler process can accept these requests in an optimal order.

- **Deadlock avoidance:** if process A sends a synchronous message to process B just as B sends a synchronous message to A, then the two processes will deadlock. However, if these processes send asynchronous messages, they will not deadlock.

3.3 *Process priorities*

Concurrent C provides facilities for specifying process priorities, querying and changing process priorities. Process priorities are specified when creating a process, but the priority can be changed at any time. Processes that need to respond quickly to events such as interrupts, which need immediate attention, are given high priorities.

3.4 *Scheduling*

Concurrent C specifies only that processes should be scheduled fairly (in the presence of priorities). An implementation can choose to implement any appropriate scheduling discipline, e.g. round robin with time slicing or preemption, as long as it is fair. In our implementations, in addition to time slicing on each processor, process interaction points (e.g. transaction calls and process creation) are used as opportunities to reschedule processes.

3.5 *Timeouts*

Concurrent C provides facilities for timed synchronous transaction calls. This allows a process to withdraw a transaction call if it has not been accepted within the specified period. A process can also call timeout if a transaction call does not arrive within a specified period. Timeouts prevent a process from being blocked for an unduly long period.

3.6 *Handling interrupts*

An important aspect of real-time programming is interrupt handling. Concurrent C provides facilities for handling interrupts in a high-level manner. The implementation-dependent function **c_associate** is used to indicate that the specified interrupt should be converted into a call to the specified transaction. To avoid losing interrupts, it is important that the associated transactions are handled quickly.

Delay in handling interrupts can occur because of the

- ☐ overhead in converting the interrupt to a transaction call;
- ☐ delay in scheduling the driver process;
- ☐ delay in accepting the associated transaction call.

These items are implementation- and application-dependent. In many cases, by giving the interrupt handling process a high priority and designing this process to give preference to accepting interrupt transactions, interrupts can be handled within real-time constraints. We have successfully used this scheme in two applications: a robot control program (discussed later) and a TTY handler.*

4 Examples

To illustrate the use of Concurrent C for real-time applications, we will write two examples, both of which handle interrupts by using the **c_associate** function to convert them to transactions.

4.1 Chain printer controller [9]

We shall illustrate the use of the **delay** alternative by writing a device driver for a chain line printer. Printer chain wear and tear is minimised by stopping the chain when the printer is not being used. If the printer is idle for 10 seconds, the driver stops the chain. The chain is restarted when a print request arrives. The printer is started and stopped by calling routines **startChain** and **stopChain**. Characters are sent to the buffer by writing to location 077502, and the printer indicates readiness to accept the next character by generating an interrupt at location 0200.

These are the specification and body of a process for controlling the printer.

File: printer.h

```
process spec printer()
{
  trans void print(char *line);
  trans void ready();
};
```

File: printer.cc

```
#include <string.h>
#include "concurrentc.h"
#include "printer.h"
void startChain(), stopChain();
void c_associate();
process body printer()
{
  char buffer[128];
  int chainRunning = 0;
  char *out = (char *) 077502;
  int i;
  process printer me = c_mypid();
  c_associate(me.ready, 0200);
  for (;;)
    select {
      accept print line)
        strcpy(buffer, line);
      if (!chainRunning) {
        startChain();
        chainRunning = 1;
      }
      for (i = 0; i < strlen(buffer); i++) {
        *out = buffer[i];
        accept ready();
      }
    or (chainRunning):
      delay 10.0; stopChain();
      chainRunning = 0;
    }
}
```

The select statement has two options: an **accept** alternative and a **delay** alternative. The latter is used to stop the chain if no **print** request (transaction call) is received within 10 seconds. Note the use of a guard expression **chainRunning** to disable the **delay** alternative when the chain is not running.

The **c_associate** function maps the interrupts to the transactions. The first argument of **c_associate** specifies the transaction to be associated with the interrupt specified by the second argument.

4.2 Robot controller [10]

The problem is to write a program to control the motion of two-dimensional (Cartesian) robots. The XY motion of the robot is provided by two orthogonal step motors. The step motor controller hardware is supplied with the direction, distance and speed of travel, and on completing the move, the motor generates an interrupt at the associated interrupt location.†

The Concurrent C program to control a single robot is structured as in Fig. 2. First, we will show you the **main** process.

```
#include <stdio.h>
#include "icma.h"
      /*contains declarations for the        */
      /*hardware functions motor_init()      */
      /*hardware_inir(), motor_complete(),*/
       - 'move() */
#include "vec.h"
#include "robot.h"
main()
{
  process robot r;
  vec pos;
  short d = 0;

  hardware_init();
  pos.x = 0; pos.y = 0;
  r = create robot(0, pos);
  r.calibrate();

  printf("type x, y: ");
  while (scanf("%d %d", &pos.x, &pos.y) == 2)
    r.move(pos);
}
```

The **main** process first calls the function **hardware_init** to initialise the hardware. Next, **main** creates the **robot**

* Private communication from Bill Roome.

† The robot program shown here was compiled on a SUN workstation and executed on the Concurrent C implementation running on a Motorola 68010-based NRTX system [11], which communicates with the robot hardware.

process and then calls its transaction **calibrate** to initialise the robot. After the initialisation, the **main** process repeatedly calls transaction **move** of the **robot** process to move the robot to user-requested positions. Note that a new move request is accepted only after the robot has completed its previous move.

File **vec.h** used in the above program defines type **vec**, which is used to specify the x and y components of a position or a velocity.

File: vec.h

```
typedef struct {
  int x, y;
} vec;
```

Below is the specification of the **robot** process.

File: robot.h

```
process spec robot(short rn, vec ip)
{
  trans void calibrate();
  trans void move(vec p);
};
```

The **robot** process takes as arguments the robot number (starting with 0) and the initial position of the robot. It has two transactions: **calibrate**, which moves the robot to the initial position, and **move**, which synchronously moves the robot to the specified position.

The body of the **robot** process is

File: robot.cc

```
#include "vec.h"
#include "robot.h"
#include "icma.h"
#include "motor.h"

process body robot(rn, ip)
{
  process motor xm, ym;
  vec cur_pos = ip, pos, v, rel_vel();

  xm = create motor(2*rn);
  ym = create motor(2*rn+1);
  for(;;)
    select {
      accept calibrate() {
        v = rel_vel(ip);
        xm.move(ip.x, v.x);
        ym.move(ip.y, v.y);
        xm.wait(); ym.wait();
      }
    or
      accept move(p) {
        pos.x = p.x. - cur_pos.x;
        pos.y = p.y - cur_pos.y;
        v = rel_vel(pos);
        xm.move(pos.x, v.x);
        ym.move(pos.y, v.y);
        xm.wait(); ym.wait();
        cur_pos = p;
      }
    }
}
```

Upon instantiation, the **robot** process immediately creates two **motor** processes. Note that motors **2*rn** and **2*rn+1** are associated with the robot numbered **rn**. The **robot** process then enters a loop in which it will accept either a **calibrate** or a **move** transaction. After accepting a **move** request, the **robot** process first computes the distances to be moved and the x and y velocities (using function **rel_vel**). Then it initiates moves in the x and y directions, by calling transaction **move** of the **motor** processes. The **robot** process then waits for the motors to finish moving, by calling transaction **wait** of the **motor** processes before updating the robot location.

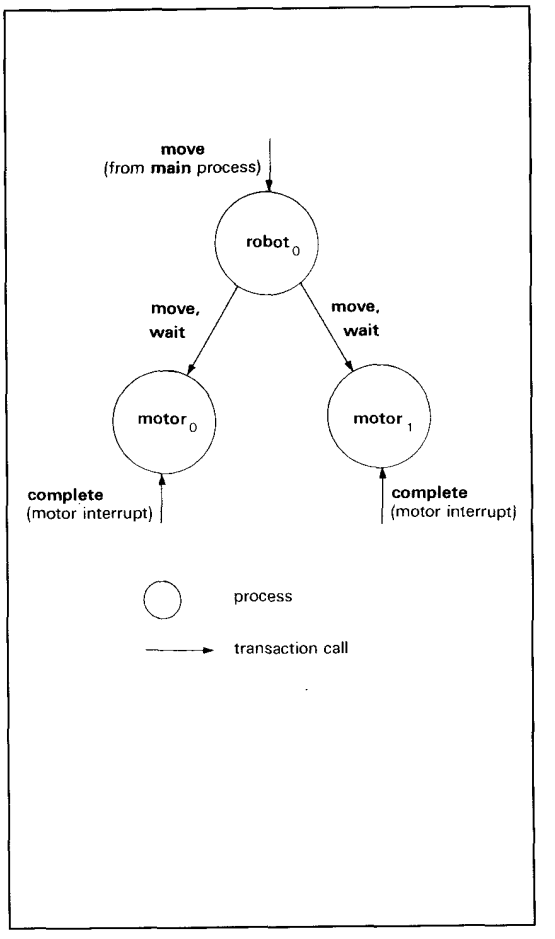

Fig. 2. Robot control program

This is the specification of the **motor** process.

File: motor.h

```
process spec motor(int motor_id)
{
  trans async move(int distance, int vel);
  trans async complete();
  trans void wait();
};
```

The **motor** process has three transactions.

- **move:** initiate the motor motion asynchronously;
- **complete:** associated with the motor interrupt, indi-

cating that the motor has completed the requested action;
- **wait:** called by the process requesting the move to wait until the move has completed.

Below is the body of the **motor** process:

File: motor.cc

```
#include "icma.h"
#include "motor.h"

process body motor(motor_id)
{
  int dist;
  trans async (*tp) ();

  tp = ((process motor) c_mypid()).complete;
  c_associate(tp, IVEC(motor_id));
  init(motor_id);
  accept complete();

  for(;;)
  {
    accept move(distance, vel)
      if (dist = distance) != 0) {
        motor_move(motor_id, distance, vel);
        accept complete()
          motor_complete(motor_id);
      }
    accept wait();
  }
}
```

After calling **c_associate,** the **motor** process initiates the hardware by calling the **init** function. The **motor** process then waits for the **complete** transaction, which indicates completion of the initialisation sequence.

The **motor** process then enters a loop in which it first accepts a **move** transaction; if the distance to be moved is greater than zero, then the function **motor_move** is called to start the motor. **motor** then waits for the completion of the motion, which is signalled by the arrival of the **complete** transaction. The **motor_complete** function call in the body of the **complete** transaction checks to see if the motor was successfully completed; if not, it prints an error message. The **motor** process then waits to accept transaction **wait** from the **robot** process before going back to the beginning of the loop.

4.2.1 Two robots in a common workspace: **suchthat** *clause example.* Modelling the single-robot program controller as a collection of interacting processes allowed us to easily extend the above program to handle two robots. To avoid collisions, **task** processes are used to manage the movements of the robots; they request permission from a manager process to move their robots and they then wait until permission is granted. After a robot has completed its move, the associated **task** process informs the manager process of this fact. We will now show you some code for the **manager** process to illustrate the use of the Concurrent C **suchthat** clause.

```
...
process body manager()
{
  for (;;)
    select {
      accept init(r, pos) { ... }
    or
      accept let_me_move(r, pos)
          suchthat(!collision(r, pos)) { ... }
    or
      accept move_done(r) { ... }
    }
}
```

Transaction **let_me_move** of the **manager** process is called by the **task** process to request permission to move. This transaction is accepted if, and only if, the move will not result in a collision. Collision avoidance is ensured by using the **suchthat** clause, which allows examination of the arguments of a transaction call without accepting the call.

5 'Hard real-time' programming

Concurrent C, like Ada, Modula 2 and other major general-purpose languages, does not provide facilities for 'hard' real-time programming. Many applications have real-time constraints which *must* be met. These languages do not provide facilities to specify and guarantee that the program will meet the specified timing constraints. It is the programmer's responsibility to ensure, by analysis and program design, that events will be handled or responded to within the desired timing constraints. A language for writing hard real-time programs must address the following issues [12, 13].

☐ **Deadline specification:** the compiler or runtime system should be able to guarantee that the specified tasks will be executed within the specified deadlines, e.g. an interrupt should be handled within the specified time. If a deadline cannot be met, then the user should be informed by means of a compile-time message or a runtime exception.

☐ **Guaranteed periodic execution:** the runtime system should be able to guarantee that some code segment, such as one sending a message to another process, will be executed periodically as specified.

☐ **Priority inversion:** it should not be the case that a high-priority process ends up waiting for a lower priority task to get service. In Concurrent C, the programmer can use the **by** clause to order requests by priority, but this must be done explicitly by the programmer.

☐ **Associate timeouts with arbitrary statements such as function calls:** currently Concurrent C allows timeouts to be associated only with process interaction statements.

☐ **Guaranteed upper bound on the suspension resulting from a process executing a delay statement:** concurrent C, like the other languages, guarantees only that the actual delay will not be less than the specified delay. No upper bound is guaranteed on the actual

delay, which is unsatisfactory from a real-time programming viewpoint.

We are currently investigating the extension of Concurrent C to support writing hard real-time programs.

6 Fault tolerance

The local area network (LAN) multiprocessor implementation of Concurrent C [14] led us to explore the design and implementation of the fault-tolerant version of Concurrent C, i.e. FT Concurrent C, which allows a program to continue operating despite the fault of some processors. Fault tolerance is particularly important for computer systems engaged in 'continuous' real-time applications, such as switching, process control, on-line databases and avionics. Even a temporary failure of one of the computers can cause breakdown of the system and could lead to disaster or require extensive work to repair the breakdown. Unfortunately, even the most reliable components are susceptible to failure. The need for fault tolerance arises if the reliability of the system is to be greater than that of its components [15]. SIFT [16] has successfully demonstrated the building of a reliable avionics system by replicating software. Replication has also been used by others to provide fault tolerance; for example, replication of complete Ada programs with tasking [17], replicated procedure calls [18], CSP processes with voting [19] and CSP processes without voting (just node failures are considered) [20]. Compared to SIFT, in which communication with the replicated processes is limited to broadcasts at the end of their activities [17], these systems provide more extensive interaction with replicated processes.

6.1 FT Concurrent C

Our goal was to extend the notion of the SIFT fault tolerance to the concurrent programming language Concurrent C. The fault-tolerant version of Concurrent C, FT Concurrent C, supports fault tolerance by allowing the programmer to replicate critical processes. A program continues to operate with full functionality, as long as at least one of the copies of a replicated process is operational and accessible. All details of interaction with replicated processes are handled by the FT Concurrent C runtime system. As far as the user is concerned, interacting with a replicated process is the same as interacting with an ordinary process. FT Concurrent C also provides facilities for notification on process termination, detecting processor failure during process interaction and automatically terminating slave processes.

An FT Concurrent C replicated process behaves like a single process. Its replicas cannot, in general, be referenced individually. Interaction with a replicated process automatically implies interaction with all the replicas. Replicas of the same replicated process are identical. However, the replicas may not exhibit the same behaviour in response to identical messages (same messages in the same order) because of the non-determinism within them and because the clocks used for the different replicas may not be synchronised. Each replica must exhibit the same external behaviour; this requires that each replica agrees about the next external event. This is accomplished, in the presence of processor failure, by using a distributed consensus protocol [21]. FT Concurrent C does all of this automatically. Note that the replicas can fall behind, as long as the FT Concurrent C implementation ensures that all the replicas exhibit the same external behaviour.

The 'death notice' mechanism allows a process to request that the FT Concurrent C runtime system generates a transaction call when (and if) a process terminates. For example, suppose a client process calls one of a server's transactions to allocate a resource, and when done, the client calls another transaction to release the resource. If the client process terminates in between, the server would not be able to reclaim the resource. The server can avoid this by asking to be notified if the client process terminates.

6.2 Replication model

In defining the semantics of interaction with the replicated processes we had two choices: to use a voting scheme [19] or to just take the response of the first replica and discard the responses of the other replicas. The 'first-response' approach* just protects against processor failures,† whereas the voting approach also masks errors resulting from some malicious processor failures that lead to erroneous responses. Although the first-response approach does not provide as much fault protection as the voting approach, it requires less redundancy (only one active replica is needed for full functionality against $2n + 1$ to mask n errors in the voting approach), and the program as a whole may run faster (because a process interacting with a replicated process does not have to wait for all the replicas to respond.§

The first-response approach was selected for FT Concurrent C. As in the Circus system [18], we may eventually allow the programmer the option of using either the voting or the first-response schemes

6.3 Programmer responsibility

The programmer must do two things: the programmer must analyse the program and decide which processes must be replicated, and the programmer must ensure that all replicas of a replicated process have the same external behaviour, i.e. if several replicas of this process are created, and if identifical input messages are presented to each replica in the same order, then each replica must generate the same output messages, in the same order. Note that replicas of a replicated process can be created with different initial arguments.

---

* Most 'check-pointing and rollback' systems also provide similar fault tolerance [22, 23].

† We assume the processors fail by stopping.

§ The replicas do not run in lock step. To be precise, although replicas can fall behind in execution, they must agree on the order in which external events take place. For example, a replica can accept a message X even though the other replicas may be lagging behind. However, before accepting X, a distributed consensus protocol is used to ensure that each replica will, when it is ready to do so, accept X. This information is stored with the other replicas. Eventually, of course, if a replica goes too far ahead, past implementation limits, then it will be suspended while the others catch up.

6.4 *Cost of fault tolerance*

Cost is an important factor in the design of fault-tolerant systems. To reduce the cost of fault tolerance, the emphasis of FT Concurrent C is on the design of fault-tolerant programs with selective fault tolerance, i.e. programs in which only the critical parts are made fault-tolerant. Besides the cost of the redundancy (i.e. replication of the processes and interacting with the replicas), the main cost is that of the consensus between the replicas. For real-time applications, hardware support may be necessary to handle the extra communications and to perform the consensus between the replicas within real-time constraints.

7 FT Concurrent C summary

7.1 *Transactions*

As in Concurrent C, processes interact by means of transaction calls. If the called process fails or does not exist, the Concurrent C runtime system terminates the program. However, the FT Concurrent C runtime system prints an error message and kills the calling process.

Termination of the calling process can be avoided by using the no-fault operator **??**; this operator allows a process to perform alternative actions if the called process terminates (e.g. because of processor failure).

transaction-call **??** *fault-expr*

If the called process accepts this call and returns a value, then that value becomes the value of the **??** operator. However, if the called process fails before accepting the call or during the transaction call, or if the called process does not exist, then *fault-expr* is evaluated and its value becomes that of the **??** expression. Note that timeouts can be associated with transaction calls.

7.2 *Death notices*

The 'death notice' mechanism allows a process to request the FT Concurrent C runtime system to call one of its transactions (the death notice) when the specified process dies. For example, suppose a client process is allocated some resource. If the client process dies before releasing the resource, the server will not be able to reclaim the resource. The server can avoid this by asking the FT Concurrent C runtime system to notify it if the client process dies. Function **c_request_death_notice** is called to request a death notice, and function **c_cancel_death_notice** is called to withdraw a death notice request.

7.3 *Creating replicated processes*

Replicated processes are created, as are ordinary processes, using the **create** operator. The newly created process can be designated as a 'slave' of the parent process. If a parent process dies abnormally, FT Concurrent C guarantees that all of its 'slave' child processes will be killed automatically. If, on the other hand, the parent process terminates normally, then its slaves are 'freed' and can run independently.

When a replicated process creates another process (replicated or not), only one instance of the process is really created. The first replica to execute the **create** operation actually creates the new process. When the other replicas execute the corresponding **create** operator, no new process is created; instead, FT Concurrent C returns the process identifier that was given to the first replica.

When creating a replicated process, the number of replicas and the processors on which these replicas should be placed are specified (the latter is optional). Process types are parameterised, and replicas can be created with different arguments. Although useful, allowing different arguments for different replicas does make it easy for the replicas to behave differently. It is the *programmer's responsibility* to ensure that the different arguments are not used to make the different replicas behave differently (see below).

7.4 *Behaviour of a replicated process*

Each replica must execute similar, if not identical code. In particular, each replica must perform the same sequence of identical 'process' interactions. For example, each replica *must* issue the same sequence of transaction calls to the same processes. Such interactions include

- transaction calls,
- accepting transaction calls,
- returning transaction results,
- waiting for a set of events,
- process creation.

Some FT Concurrent C statements are non-deterministic, i.e. execution of the same statement may lead to different results. For example, the **select** statement, which is used to wait for one of a set of events, is non-deterministic. FT Concurrent C ensures, by using a 'distributed consensus' protocol [21], it makes the same choice for each replica. There is no requirement for the replicas to operate in lockstep. FT Concurrent C ensures, by storing appropriate information, that each replica performs the same action as its siblings, who might be ahead in executing their code.

7.5 *Calling a replicated process*

When a replicated process is called, FT Concurrent C automatically sends the call to all replicas of the replicated process. When the first replica completes the call, FT Concurrent C activates the calling process and gives it the value returned by that replica. The other replicas will eventually accept this transaction and generate replies, but FT Concurrent C will automatically discard the values returned by them. If a replica terminates because of a processor failure, FT Concurrent C automatically deletes it from the set of replicas, without informing the caller.

7.6 *Transaction calls from a replicated process*

A transaction call made by a replicated process means that

each replica should generate an identical transaction call. However, only the first of these transaction calls will actually be executed by the called process. FT Concurrent C makes sure that each replica gets the same transaction result. In case of a timed-transaction call (transaction call with a timeout) or a no-fault (**??**) transaction call, FT Concurrent C ensures, by using the distributed consensus protocol, that each replica takes the same alternative.

7.7 Accept and return statements in replicated process

When one replica executes an **accept** statement, FT Concurrent C ensures that the other replicas will accept, or have accepted, the same transaction call. If the **accept** statement has **suchthat** and **by** clauses, these clauses are evaluated only when first deciding which transaction call to accept. Once this decision has been made, other replicas must accept the same transaction call, regardless of the values of the **suchthat** and **by** expressions. Only one replica need send the transaction result to the calling process. The results of the other replicas may be simply discarded by the FT Concurrent C implementation.

7.8 Select statements in replicated processes

FT Concurrent C ensures each replica will take the same alternative. In the case of **accept** alternatives, FT Concurrent C ensures that identical transaction calls are accepted by each replica. Side-effects should be avoided in the expressions used in the **select** statement guards, and in the **by** and **suchthat** clauses, because these expressions may, or may not, be evaluated in all the replicas.

7.9 FT Concurrent C example: the dining philosophers [24]

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** ($0 \leqslant i \leqslant 4$). Five forks have also been laid out on the table, so that there is precisely one fork between every two adjacent 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 him/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 he/she must wait until he/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), he/she puts the forks down and leaves the room.

In the straightforward solution, the processes representing the **philosophers** and **forks** can all be replicated. To reduce the redundancy, only the critical processes can be replicated. In the fault-tolerant version of the dining philosophers shown below, only the **forks** are replicated (two copies), and death notices are used to find out when a **philosopher** process terminates without releasing the shared resource (the fork).

File: ftdin.cc

```
#define LIMIT 100000
process spec fork()
{
  trans void pick_up(), put_down();
  trans void died(process anytype);
};
process spec philosopher(int id, process fork left,
                                  process fork right);
process body philosopher(id, left, right)
{ int times_eaten;
  for (times_eaten = 0; times_eaten != LIMIT;
                        times_eaten++) {
    /*pick up forks*/
      right.pick_up(); left.pick_up();
    /*eat*/
      printf("Philosopher %d: That was deliciousTn", id);
    /*put down forks*/
      left.put_down(); right.put_down();
  }
}


process body fork()
{ process anytype p;
  trans void (*tp) () = ((process fork)c_mypid()).died;

  for (;;) {
    select {
      accept pick_up()
        p = c_caller_pid();
    or
      terminate;
    }
    c_request_death_notice(p, tp);
    select {
      accept put_down();
      c_cancel_death_notice(p, tp);
    or
      accept died(p);
    }
  }
}


main()
{ process fork f[5]; int j;
  process (j=0; j<5; j++)
    f[j] = create fork() copies(2);
  for (j = 0; j<5; j++)
    create philosopher(j, f[j], f[(j+1)%5]);
}
```

If the **main** process terminates prematurely after creating the **fork** processes, i.e. before creating the **philosopher** processes, then these processes will automatically terminate by executing the **terminate** alternative [2].† Note that the **fork** process requests a death notice for the **philosopher** process to which it gives a fork. This request is cancelled when the fork is returned.

* This solution can lead to a deadlock.

† FT Concurrent C does not, at present, support replication of the **main** process, i.e. the initial process. It is expected that this process will be small and that it will be used to create replicated processes which make the program fault-tolerant.

8 Summary

Concurrent C extends the C language by providing facilities for concurrent programming. Although Concurrent C and Ada are both based on the rendezvous model, there are substantial differences between the facilities provided by the two languages. For example, Concurrent C, unlike Ada, provides facilities for asynchronous message passing and for accepting messages in a user-specified order [25]. Concurrent C has been in use for four years now. Several implementations for different hardware configurations are now available. Except for the addition of facilities for running programs on multiprocessors and asynchronous message passing, the language has been quite stable. Although Concurrent C supports the writing of real-time programs, it does not provide facilities to specify timing constraints, such as the specification of the maximum delay, which should occur in handling an event. As mentioned earlier, it is the programmer's responsibility to ensure, by analysis and program design, that events will be handled or responded to within the desired timing constraints. We are currently investigating how to extend Concurrent C with facilities to support the writing of hard real-time programs.

FT Concurrent C is a tool for writing fault-tolerant distributed systems. The use of replication to handle processor failures is appropriate for systems that require continuous operation, e.g. industrial processes [26]. As cost is a significant factor in writing fault-tolerant systems, FT Concurrent C provides the programmer the ability to specify selective fault tolerance. For some real-time applications, hardware support, in the form of a communications processor, which handles the extra communications and the consensus between the replicas, may be appropriate or necessary.

We have implemented a prototype version of FT Concurrent C. Whenever replicated processes are involved, the underlying runtime system uses a distributed consensus protocol to ensure that the replicas take the same action. The implementation's responsibility in this extends only to concurrency operations (transaction calls, **select** statements, process creation etc.). One question we have not yet addressed is the bringing of failed processors back on-line after they have been restored or replaced.

9 References

[1] GEHANI, N.H.: '(Fault Tolerant) Concurrent C'. Proc. of IFIP/IFAC Conf. on Hardware and Software for Real Time Process Control, Warsaw, Poland, June 1988, ZALEWSKI, J., and EHRENBERGER, W. (Eds.) (North-Holland), pp. 87–100

[2] GEHANI, N.H., and ROOME, W.D.: 'Concurrent C' (Silicon Press, 1989)

[3] STROUSTRUP, B.: 'The C++ programming language' (Addison Wesley, 1986)

[4] CMELIK, R.F., GEHANI, N.H., and ROOME, W.D.: 'Fault tolerant concurrent C'. 18th Int. Symp. on Fault-Tolerant Computing, June 1988, Tokyo, Japan, pp. 56–61

[5] BLOOM, T.: 'Evaluating synchronization mechanisms'. Proc. Seventh Symp. on Operating Systems Principles, ACM-SIGOPS, December 1979, pp. 24–32

[6] LISKOV, B., HERLIHY, M., and GILBERT, M.: 'Limitations of synchronous communication with static process structure in languages for distributed computing'. Proc. 13th ACM Symp. on Principles of Programming Languages, St. Petersburg, Florida, 1986, pp. 150–159

[7] GEHANI, N.H.: 'Message passing: synchronous vs asynchronous', *Softw. — Pract. Exp.*, 1990, **20,** (6), pp. 571–592

[8] ANDREWS, G.R., and OLSSON, R.A.: 'The evolution of the SR language', *Distrib. Comput.*, 1986, **1,** (3), pp. 133–149

[9] Department of Defense: 'Rationale for the design of the Ada programming language', *SIGPLAN Not.*, 1979, **14,** (6), Part B

[10] COX, I.J., and GEHANI, N.: 'Concurrent C and robotics'. IEEE Conf. on Robotics and Automation, Raleigh, North Carolina, 1987, pp. 1463–1468

[11] KAPILOW, D.A.: 'Real-time programming in a UNIX environment'. Symp. on Factory Automation and Robotics, New York University, USA, 1985, pp. 28–29

[12] BAKER, T.: 'Fixing some time-related problems in Ada'. 3rd Int. Workshop on Real-Time Ada Issues, June 1989, Pittsburgh, Pennsylvania, pp. 63–70

[13] WREGE, D.E.: 'Real-time Ada issues in the Ada language system/Navy'. 3rd Int. Workshop on Real-Time Ada Issues, Pittsburgh, Pennsylvania, June, 1989, pp. 131–136

[14] CMELIK, R.F., GEHANI, N.H., and ROOME, W.D.: 'Experience with multiple processor versions of concurrent C', *IEEE Trans.*, 1989, **15,** (3), pp. 335–344

[15] DRUMMOND, R.: 'Impact of communication networks on fault tolerant distributed computing'. PhD Thesis, TR86–748, Cornell University, New York, 1986

[16] WENSLEY, J.H., LAMPORT, L., GOLDBERG, J., GREEN, M.W., LEVITT, K.M., MELLIAR-SMITH, P.M., SHOSTAK, R.E., and WEINSTOCK, C.B.: 'SIFT: design and analysis of a fault-tolerant computer for aircraft control', *Proc. IEEE*, 1978, **66,** (10), pp. 1240–1254

[17] MELLIAR-SMITH, P.M., and SCHWARZ, R.L.: 'A fault-tolerant Ada architecture'. Proc. 4th Jerusalem Conf. on Information Technology, Jerusalem, Israel, 1984, pp. 211–215

[18] COOPER, E.C.: 'Replicated distributed programs'. Proc. 10th ACM Symp. on Operating System Principles, *Oper. Syst. Rev.*, 1985, **19,** (5), pp. 63–78

[19] MANCINI, L.: 'Modular redundancy in a message passing system', *IEEE Trans.*, 1986, **SE-12,** (1), pp. 79–86

[20] JALOTE, P., and TRIPATHI, S.K.: 'Fault tolerant computation in synchronous message passing systems'. Report, University of Maryland, USA, 1986

[21] AREVALO, S., and GEHANI, N.: 'Replica consensus in fault tolerant concurrent C'. AT&T Bell Laboratories, 1989

[22] LISKOV, B., and SCHEIFLER, R.: 'Guardians and actions: linguistic support for robust, distributed programming', *ACM TOPLAS*, 1983, **5,** (3), pp. 381–404

[23] SVOBODOVA, L.: 'Resilient distributed computing', *IEEE Trans.*, 1984, **SE-10,** (3), pp. 257–268

[24] DIJKSTRA, E.W.: 'Hierarchical ordering of sequential processes', *Acta Inform.*, 1971, **1,** pp. 115–138

[25] GEHANI, N.H., and ROOME, W.D.: 'Rendezvous facilities: concurrent C and the Ada language', *IEEE Trans.*, 1988, **14,** (11), pp. 1546–1553

[26] WEBER, M.: 'Operating system enhancements for a fault-tolerant dual-processor for the control of an industrial process', *Softw. — Pract. Exp.*, 1987, **17,** (5), pp. 35–350

The author is with AT&T Bell Laboratories, 600 Mountain Avenue, Room 3D-414, Murray Hill, New Jersey 07974, USA.

The paper was first received on 15th December 1989 and in revised form on 4th April 1990