CELL: A DISTRIBUTED COMPUTING MODULARIZATION CONCEPT

Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-155 May, 1983

CELL: A DISTRIBUTED COMPUTING MODULARIZATION CONCEPT[1]

Abraham Silberschatz
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
512-471-4353

Abstract

This paper presents a new language construct for distributed computing. This construct, called cell, allows one to simulate a variety of language constructs. Its salient features provide the programmer with:

a) an effective communication and synchronization scheme,

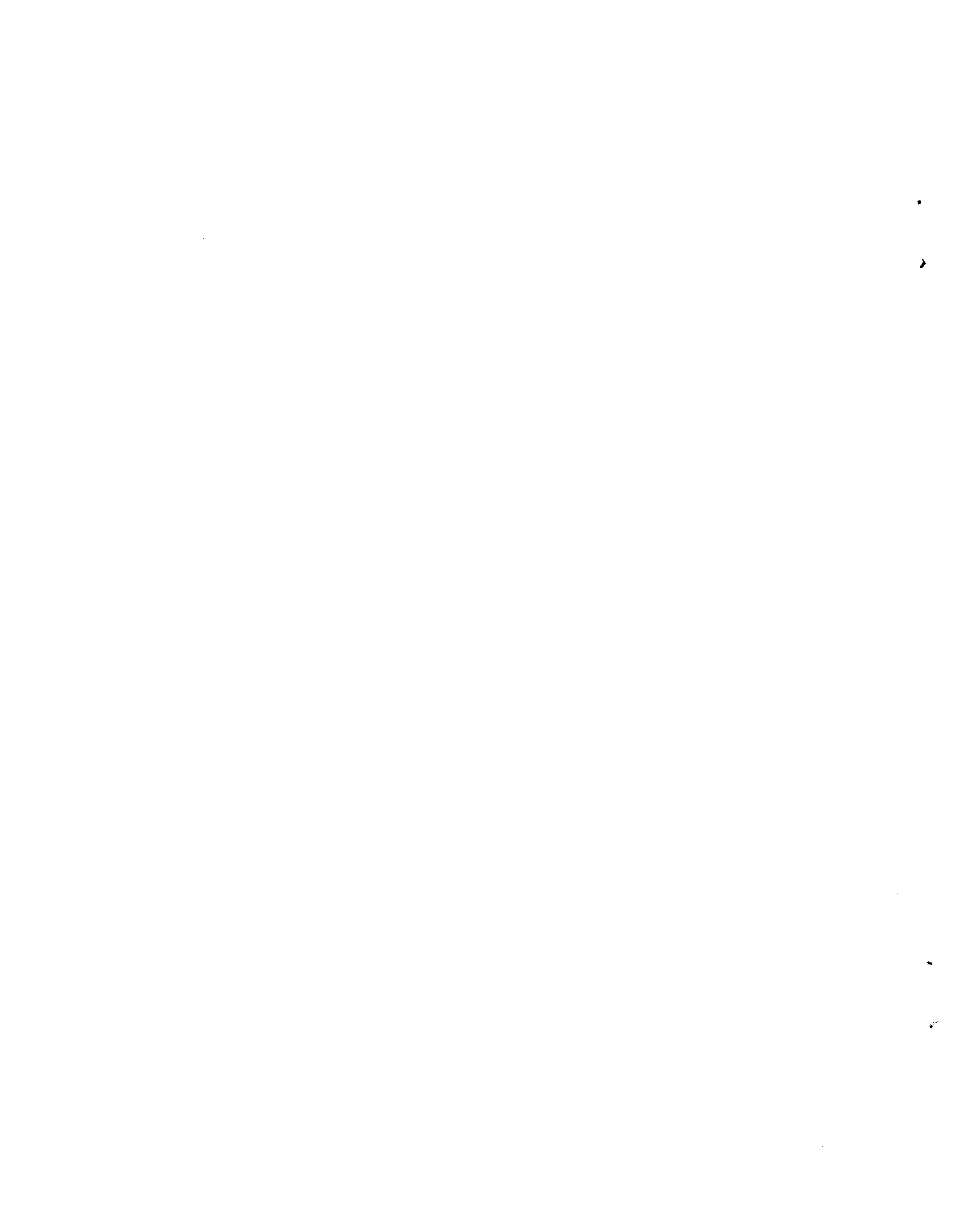
b) a mechanism to control the order in which various activities within a cell should be executed.

We demonstrate the usefulness of our concepts by providing solutions to a variety of programming exercises.

Index Terms -- communication, distributed systems, programming languages, scheduling, synchronization.

---

[1]This research was supported in part by the National Science Foundations under Grant MCS 8104017, and in part by the Office of Naval Research under Contract N00014-80-K-0987



1. Introduction

Recent advances in technology have made the construction of general purpose systems out of many small independent micro-processors feasible. One of the issues concerning distributed systems is the question of appropriate language constructs. When the functions of a system are distributed over many independent micro-processors, many of the well established ideas in system design, language specification, synchronization and communication do not directly apply to such systems. As a result several new language concepts have been recently proposed to overcome these difficulties. A partial list includes Ada [1], Communicating Sequential Processes [2], Distributed Processes [3], E-Clu [4], Gypsy [5], *MOD [6], PLITS [7], and Synchronizing Resources [8].

The cell concept presented in this paper is yet another entry to this list. It was developed because of the shortcomings of the various language constructs mentioned above. The cell construct provides a programmer with an effective and efficient synchronization scheme. It also provides the programmer with a mechanism to control the order in which various activities within a cell should be executed. This is of the utmost importance when one writes distributed programs.

No claim is made that the cell concept does not overlap ideas with other similar previously developed concepts. While presenting our results we do however argue, that it possesses most of the good features of the other similar concepts, few of the bad features, and some advantages of its own. Illustrative examples to demonstrate the versatility of the cell concept will also be presented.

The remainder of the paper is organized as follows. Section 2 presents the basic structure of the cell construct. In Section 3 we describe two new language features that allow one to efficiently and effectively implement a variety of synchronization schemes. Finally, Section 4 presents the overall structure of our scheme, illustrating how a distributed program can be constructed out of a number of individual cells.

2. The Cell Construct

A distributed program consists of a finite number of modules, called cells, that are executed concurrently. In this section we present the basic structure of the cell construct. Since we are not interested in presenting a complete language we illustrate our concepts using a Pascal-like notation [9]. Thus a distributed program may be viewed as a program written in a Pascal based language augmented by the

cell construct. The cell module has the following form[2]:

```
<identifier>: cell [<permanent parameters>]
    [<declaration>]
  begin <statement> end.
```

A cell can only access its own variables; that is, the permanent parameters, and those variables defined in the declaration part. The permanent parameters are passed during the initialization and termination phase of a cell. We shall return to the question of how permanent parameters are handled in our scheme in Section 4.

Cells communicate and synchronize with each other via a variant of the accept and select statements of Ada [1]. The accept statement has the following form:

```
accept <entry-name> [<formal parameter list>]
   [do <statements> end;]
```

The <statements> of an accept statement can be executed only if another task invokes the entry-name. Invoking an entry-name is syntactically the same as a procedure call. At this point, parameters are also passed. After the end statement has been reached, parameters may be passed back, and both tasks are free to continue. Either the calling task or the called task may be suspended until the other task is ready with its corresponding communication. Thus the facility serves both as a communication mechanism and a synchronization tool.

In order to make communication between the calling and called cells symmetric with respect to naming, a mechanism is provided to allow the cell to selectively choose among those cells with whom communication is to take place. This is done by extending the syntax of the accept statement to include an optional clause,

from <cell>

where <cell> is either a name of a cell, or a variable of type identity -- an enumeration type which consists of all possible cell names. The from clause may appear after the formal parameter list of an accept statement. In order for our scheme to work, a mechanism must exist to allow the programmer to dynamically determine the identity of various cells. For this purpose we postulate a language defined function, caller, which can be used only in an accept statement, and which returns the name of the calling cell.

---

[2]Square brackets "[ ]" denote an optional part, while braces "{ }" denote a repetition of zero or more times.

Since cells are completely disjoint in logical address space, parameters may be passed either by value (in parameters), or by result (out parameters). Call by reference is disallowed since we do not assume a shared memory architecture. For the same reason pointer-type variables may not be passed as actual parameters. Note that our in/out parameters are semantically different from those of Ada since we specify precisely how they are to be passed (implemented).

Choices among several entry calls is accomplished by the select statement, which is based on Dijkstra's guarded command concept [10]. The select statement has the form:

```
select
   [when <boolean-expression> => ]
       <accept-statement>
        [<statements>]
   {or [when <boolean-expressions> => ]
       <accept-statement>}
        [<statements>]
   [else <statements>]
end select;
```

Execution of a select statement proceeds as follows:

1. All the boolean-expressions appearing in the select statement are evaluated. Each accept statement whose corresponding boolean expression is evaluated to be true is tagged as ready. An accept statement that is not preceded by a when clause is always tagged as ready.
2. A ready accept statement may be selected for execution only if another cell has invoked an entry corresponding to that accept statement. If several ready statements may be selected, an arbitrary one will be chosen for execution. If none can be selected and there is an else part, the else part is executed. If there is no else part, then the cell waits until a ready statement can be selected.
3. If no accept statement is ready, and there is an else part, the else part is executed. Otherwise an exception condition is raised.

The accept statement provides a task with a mechanism to wait for a predetermined event in another task. On the other hand, the select statement provides a task with a mechanism to wait for a set of events whose order cannot be predicted in advance.

These concepts can be best illustrated by sketching out a simple Printer cell, whose function is to control a single physical printing device. A user who wishes to output a sequence of lines to the printer simply sends these lines to the Printer cell, terminating this sequence with the character string "EOL". Obviously lines from several different users should not be mixed.

```
Printer: cell;
   var Buffer: Line;
       Current: identity;
   begin
      repeat
         accept Put_line (L: Line) do
            Buffer: = L;
            Current: = caller;
            PRINT (Buffer);
         end
         while (Buffer<> 'EOL') do
         begin
            accept Put_line (L: Line) from Current do
               Buffer: = L;
            end;
            PRINT (Buffer);
         end;
      until false;
   end.
```

PRINT is an unspecified program segment corresponding to the actual printing of a line.

3. Scheduling

A major area of application where the cell construct can be utilized is resource scheduling. In such a scheme a scheduler cell must be provided whose function is to coordinate orderly access to a resource. Such a scheduler accepts calls from customer cells and processes them in some fashion. Although it is possible to implement such a scheme using the basic cell construct, it cannot be accomplished in a straightforward, concise and efficient manner. In this section we propose two new language constructs that can be effectively used in implementing a variety of resource scheduling schemes.

3.1. The Await Statement

In order to implement resource scheduling schemes, a mechanism must be available to delay a customer process before an access permission can be granted. The vast majority of standard resource scheduling examples in the literature can be cleanly implemented by delaying a customer cell just once in a static priority-ordered queue. Moreover, some code may have to be executed before the scheduler knows which queue and what priority is appropriate. In order to effectively deal with this issue, we propose a new queuing mechanism, that allows cells to delay calls after they have been accepted. This mechanism is a variant of Kessels conditional wait construct [11]. Central to this mechanism is the command:

await (<boolean-expression>)

The await statement can appear only in an entry operation associated with a select

statement.

With each await statement a list (called an await-list) is associated consisting of entries (activation records) each of which contains:

a. the local variables of the accept statement, where the await is defined.

b. information concerning the calling cell.

c. address of the statement following the await statement.

An activation record is used to save a partial state of a computation so that this state can be restored at a later point in time.

When a cell encounters an await statement it evaluates the boolean-expression associated with this statement. If true, the cell continues with its execution. If false, the cell obeys the following:

i. it creates a new activation record, initializes it, and adds it to the appropriate await-list.

ii. it proceeds with its execution at the first statement following the select statement in which the await is defined.

Thus an await statement provides the programmer with a mechanism to switch between various distinct computations.

With this new mechanism we have to redefine the manner in which the execution of a select statement proceeds. As before, all the boolean expressions within the select statements are first evaluated. This time however, the boolean expressions of the await statements defined within the select statement, are also included. Each statement and activation record whose associated boolean expression is evaluated to be true is tagged as open. If no statement or record is open, the execution proceeds as before. If several statements or records are open, an arbitrary single one is selected. If it is a statement then it is simply executed. If it is record, then it is removed from the associated await-list and the state of the cell is restored using the information from the removed record. Note that if a state restoration occurs, the value of the program counter is also being affected.

In order to give the programmer a closer control over scheduling, we extend the await statement to allow the inclusion of priority information [12]:

<u>await</u> (<boolean expression>) [<u>by</u> (<priority-expression>)]

The <u>by</u> clause is optional. When a cell encounters an await statement with a false boolean-expression, then the priority-expression is evaluated to produce an integer priority value. This priority value is also stored in the activation record added to the await-list. When an activation record is picked during the execution of a select

statement, the activation record with the smallest priority value is removed. If no priority value is present an arbitrary record is removed. This new feature allows the coding of many common synchronization problems (e.g. shortest-job-next, alarm clock, disk schedulers) with the boolean expression restricted to range only over the cell's own variables.

Let us illustrate these concepts by considering a simple resource scheduling scheme. Suppose that one wishes to define a cell whose function is to allocate a resource among a number of cells in the shortest job next order:

```
SJN: cell
   var Free: boolean;
   begin
      Free:= true;
      repeat
         select
            accept Acquire (Time: in integer) do
               await (Free) by (Time);
               Free:= false;
            end;
         or
            accept Release;
            Free:= true;
         end;
      until false;
   end.
```

Let us consider now a more complicated example of an interprocess communication (IPC) scheme [13]. Suppose many cells must occasionally communicate via message buffers, where the messages are addressed by a rendezvous code. Since the producer-consumer dialogs are infrequent, dynamic allocation of message buffers from a pool is therefore appropriate. A cell needing to communicate calls the IPC allocator cell with a rendezvous code to obtain a message buffer (Open), writes one or more messages to the buffer (Send), and releases the buffer (Close). Another cell capable of handling messages with that rendezvous code would call the allocator with the same rendezvous code (Open), read messages (Receive), and then release the buffer (Close).

We assume that there are 10 message buffers to be allocated among customer cells. For brevity we only present those program segments that illustrate our concepts. The main data structures needed are:

```
Resource:  array [1..10] of record
                                Rendezvous_code: integer;
                                Hold_count: integer;
                            end
Count, Last_index, Last_channel: integer;
```

Central to the IPC allocator is the select statement described below whose function is to accept two types of calls with formal parameters: Channel_id – the

rendezvous code supplied by the caller, and Buffer_id -- the index corresponding to the allocated message-buffer.

```
select
    accept Open(Channel_id: integer; Buffer_id: out integer) do
        var Index: 1..10;
        for Index:= 1 to 10 do with Resource[Index] do
            if Rendezvous_code = Channel_id then
                begin
                    Hold_count:= succ(Hold_count);
                    Buffer_id:= Index;
                    return;
                end
        await(Count < 10 or Channel_id = Last_channel);
            if Count < 10 then
                for Index:= 1 to 10 do with Resource[Index] do
                    if Hold_count = 0 then
                        begin
                            Rendezvous_code:= Channel_id;
                            Hold_count:= 1;
                            Buffer_id:= Index;
                            Last_index:= Index;
                            Last_channel:= Channel_id;
                            Count:= succ(Count);
                            return;
                        end
            else
                begin
                    with  Resource[Last_index] do
                    Hold_count:= succ(Hold_count);
                    Buffer_id:= Last_index;
                end;
    end;
or
    accept  Close(Buffer_id: integer) do
        with  Resource[Buffer_id] do
            Hold_count:= pred(Hold_count);
            if Hold_count = 0 then
                begin
                    Rendezvous_code := 0;
                    Count:= Count-1;
                end;
    end;
end;
```

Note that a considerable amount of code needs to be executed before the await statement is encountered. Again, we encourage the interested readers to try and code this scheme; this time however, in either Ada or SR. We also note that in the above example our program does not ensure that a cell waiting to open an IPC channel will not be overtaken by a newly arriving cell. This deficiency could be remedied by incorporating appropriate boolean expressions (using the when clause) within the select statement. More on this subject in the next section.

The above two examples were specifically chosen to illustrate the versatility of our construct, and to allow us to compare our proposal with some of the other proposed languages. We contend that the shortest job next scheme could not be

coded in a straightforward and concise manner in either Ada or DP. This is because neither one of the languages provide an effective queuing mechanism. We also contend that the IPC scheme could not be simply coded in either ADA or SR. This is due to the fact that in these languages a process (task) can be effectively delayed only at its initial entry to another module.

We conclude this section by pointing out that the await statement was specifically designed so that it can be implemented efficiently. If the boolean expressions range only over the cell's own variables then the number of evaluations needed to perform a specific task is minimal. We have introduced the by clause as an extension to the await statement so that a larger number of synchronization schemes could be coded with the boolean expression restricted to own variables only. Additional saving can be obtained by resorting to various optimization techniques (e.g. Schmid [14]). For example, in the SJN cell described above the await statement needs to be evaluated only after a Release cell has been accepted.

3.2. Ordering Specification

At any given instant there may be a number of activities ready to execute within a cell. These include the various statements within a select statement, as well as older activities ready to be dequeued from await statements. Different scheduling constructs allow different possibilities, but in all cases there can be many ready activities. Although nondeterminacy is both desirable and inevitable, the language designer must be careful to allow the programmer enough control to implement his own scheduling policies.

One way to achieve such a control is to add more information in the various boolean expressions of the select and await statements. For this purpose the count attribute of Ada is very useful. The count attribute allows the programmer to specify priorities among the various activities. However, when such a selection is required, it becomes very tedious and cumbersome to state such priorities. This is because the number of components (and count attributes) in each boolean expression increases proportionally to the number of activities among which selection is required.

Let us thus propose another approach for specifying priorities among the various ready to run activities. We provide the programmer with a specification notation to define a partial order among such possible activities. The syntax we have chosen is:

order (L1 < L2; L3 < L4; ... ; Lm-1 < Lm)

where each Li is a label attached to either an accept statement (or its associated when statement), or an await statement.

The order list specifies an order in which activities should be processed. For example, consider the following statement:

order $(A < B; A < C)$;

suppose that at some point in time the boolean expression associated with the statements labeled A,B and C are all evaluated to be true. Since $A < B$ and $A < C$ the statement labeled A will be executed first. If at some later time, the statements labeled B and C can be executed, either one of them can be selected; the choice is up to the implementation.

We note that the order list concept can be effectively and efficiently implemented. The compiler needs to first verify that the partial order among the various activities (labels) indeed holds (if not a syntax error will be recorded); this can be checked in polynomial time. Once this has been verified the compiler simply generates code to evaluate the various boolean expressions in the proper order.

With this new mechanism one could rewrite the SJN cell as follows:

```
SJN: cell;
   order (L3 < L2; L2 < L1);
   var  Free: boolean;
   begin
      Free:= true;
      repeat
         select
            L1: accept Acquire (Time: in integer) do
                   L2: await (Free) by (Time);
                       Free:= false;
                end;
         or
            L3: accept Release;
                Free:= true;
         end;
      until false;
   end.
```

This version of the SJN cell is potentially more efficient than its predecessor because the compiler is provided with information concerning the proper order of evaluation of the various alternatives within the select statement.

As a final example, consider the IPC scheme presented in the previous section. As noted, the program does not ensure that the IPC channels are allocated in a "fair" manner. This could be easily taken care off using our order list concept, by giving the "close" guard priority over the await statement, and the await statement priority over the "open" guard.

4. Overall Structure

Up to now we have only presented the central features of the cell construct. We wish to shift our attention now and present the overall structure of the cell, as well as the overall structure of a distributed program.

A programmer may define a single instance of a cell type (as has been done in the previous section), or, a general type of which several copies may exist. The latter can be accomplished through a definition of the form:

```
type <identifier> = cell[<permanent parameters>]
    [<declarations>]
   begin <statements> end.
```

The permanent-parameters are the usual in/out parameters, as defined in Section 2. A programmer may declare an instance of a previously defined cell type using the syntax:

var <identifier> : <cell_type>;

This declaration does not result in the allocation of space for the cell's own variables. This function is only accomplished during the initialization phase through the init statement which has the form:

init <identifier>[<actual-permanent-parameters>]

The init statement allocates space for the cell's own variables, passes the actual in parameters, and starts the execution of the named cell instance. At this point in time, execution also proceeds at the statement following the init statement. When a cell reaches its last statement, its out permanent parameters are copied back to the address space of the father, and it then terminates. A cell can exit a block, in which a number of cells have been declared, only when all these declared cells have terminated.

For convenience, we add one more feature to the language to allow a cell to initialize another cell and to immediately wait for that cell to terminate. This can be accomplished through the call statement which has the form:

call <identifier>[<actual-permanent-parameters>]

Thus the "caller" is delayed until the newly created cell reaches its last statement and copies the out parameters (if any) back.

Thus, we have established here a dynamic hierarchical structure, where the root of the hierarchy is the main program, and the rest of the hierarchy consists of active cells.

The above described facility allows one to simulate a variety of language features. For example, a Pascal procedure that finds the largest integer in an array of 100 elements can be simply constructed as follows:

```
Max: cell(A: in Int_Array_type; Large: out integer);
   var I: 2..100;
   begin
      Large:= A[1];
      for I:= 2 to 100 do
         if Large < A[I] then Large:= A[I];
      end;
   end.
```

A user can determine the largest number in an array B via,

call MAX(B,L);

Note that the "caller" cell is delayed until MAX terminates.

Recursive procedures can also be simply handled via our constructs using the scheme sketched below.

```
type Rec_Proc = cell(...);
           var R: Rec_Proc;
               ...
           begin
               ...
              call R(...);
               ...
           end.
```

A user can invoke the above "procedure" by declaring

var P : Rec_Proc;

and "calling" P via

call P(...);

Note that our scheme works, since the declaration "var R: Rec_Proc" (within the Rec_proc type), does not create a new cell; this occurs only when the statement "call R(...)" is executed.

One nice consequence of the hierarchical structure of our scheme is that in situations where one cell requires the service of another (e.g. procedures, monitors), the actual service may be either performed solely by the service cell, or in parallel by several other cells defined within the service cell. The details of implementation are hidden from the customer cell. For example, one may design a service cell, Sort, whose function is to sort an array of 100 elements. The Sort cell may either use a sequential algorithm (e.g., bubble sort), or, a parallel algorithm (e.g., a parallel bucket sort [15]). A cell requiring the service of the Sort cell need not be aware which algorithm is used.

To illustrate this point, consider the sort cell described below, whose function is to sort an array of N integers in time O(N). The sort cell uses an array of N cells to perform its function. Each of these cells is an instance of a type,

```
type Min = cell (M: in Min; Count: in integer;
                 Small: out integer);
   var T,Temp: integer;
   begin
      accept Put (C: integer) do Small:= C end;
      while (Count > 0) do
         begin
            accept Put (C: integer) do Temp:= C end;
            if Temp < Small then
               begin
                  T:= Small;
                  Small:= Temp;
                  Temp:= T;
               end;
            M.Put (Temp);
            Count:= Count - 1;
         end;
   end.
```

The Sort cell can now be described.

```
Sort: cell (A: in out Int_array_type);
   type Min= cell ( )...
   const N = 100;
   var Mn: array [0..N-1] of Min;
   begin
      for i:= N-1 downto 0 do
         begin
            init Mn[i] (Mn[i+1 mod N],(N-i-1),A[i]);
         end;
      for i:= 0 to N-1 do
         begin
           Mn[0].Put(A[i]);
         end;
   end.
```

Note that when Sort terminates the array A is sorted and is copied back to the address space of the "caller".

Interested readers should compare our solution with the one presented in [3]. They will find our solution to be more efficient. This is due to the fact that communication between cells can occur not only during the normal course of execution but also at initialization/termination time.

As a final example, we present a solution to the dining philosopher problem [2]. This will allow us to illustrate how a complete distributed program can be developed using our proposed scheme. Each philosopher must enter a room before he is allowed to pick up the needed forks. In order to prevent deadlocks only N-1 philosophers are allowed to be in the room at the same time.

```
type Room = cell (N: in integer);
   var Count: integer;
   begin
      Count:= 0;
      repeat
         select
            when (Count < N) => accept Enter;
                                Count:= Count + 1;
         or
            accept Exit;
            Count:= Count - 1;
         end;
      until false;
   end.
```

Each fork is picked up and put down by a philosopher sitting on either side of it.

```
type Fork = cell;
   begin
      repeat
         accept Pickup;
         accept Putdown;
      until false;
   end.
```

Each Philosopher eats only after he has successfully entered the room and has picked up the left and right forks (in that order). After eating, the left and right forks are put down and the philosopher exits the room.

```
type Phil = cell (Left,Right: Fork; R: Room);
   begin
      repeat
         THINK;
         R.Enter;
         Left.Pickup; Right.Pickup;
         EAT;
         Left.Putdown; Right.Putdown;
         R.Exit;
      until false;
   end.
```

Finally, the entire scheme could be put together,

```
Dining_Phil : program;
   type Room . . .
        Fork . . .
        Phil . . .
   const N = 50;
   var Rm: Room;
       Fk: array [0..N-1] of Fork;
       Ph: array [0..N-1] of Phil;
   begin
      init Rm;
      for i:= 0 to N-1 do;
         init Fk(i);
      end;
      for i:= 0 to N-1 do;
         init Ph(Fk(i), Fk(i+1 mod N),Rm);
      end;
   end.
```

Note that in the above example a Phil cell (e.g., Ph[ ]) gets to know the identity of the left and right fork handler cells only at initialization time. Thus our permanent parameter facility also provides the needed mechanism for establishing static access binding. Also note that these bindings cannot be modified by the Phil cell.

5. Conclusion

We have presented a new language construct called cell, that can be used as a building block in a high level language for distributed computing. The cell construct was developed because of the shortcomings of the various language constructs recently proposed in the literature. In particular, the await statement was proposed in order to provide the programmer with an effective and efficient synchronization scheme; the order list was introduced to allow the programmer to specify priorities among the various ready to run activities within a single cell; the permanent-parameter passing facility was introduced to allow cells to communicate during normal execution time as well as during initialization and termination. We have shown how these constructs could be utilized in the writing of various programming exercises. These exercises were specifically chosen to illustrate the differences between the cell constructs and other previously proposed languages. Hopefully we have convinced the readers that the aforementioned languages indeed have some severe shortcomings some of which were taken care of by our proposed construct.

We have not however shown that the cell construct does not introduce new problems, nor have we shown that it is not inferior (in some other problem domains) to some of the constructs we have criticized. These two questions cannot be easily answered. We have coded a variety of programming exercises using our language as well as some other languages and have found the cell to perform as well (or better) than the other constructs. This obviously does not establish the superiority of the cell construct but this has convinced us that in many circumstances this is indeed the case.

We have shown, through examples, that the cell construct allows one to simulate a variety of language features (e.g., procedures, recursive procedures, monitors). It can be easily shown that other language constructs (e.g., coroutines, classes, path expressions, semaphores) can also be simulated.

The work we have presented is only a first step toward the development of a complete distributed programming language. Further work needs to be done on such important issues as proof rules, implementation, exception handling, etc.

References

[1] "Reference Manual for the Ada Programming Language," United States Department of Defense, July 1980.

[2] C.A.R. Hoare, "Communicating Sequential Processes," CACM, vol. 21, no. 9, pp. 666-677, August 1978.

[3] P. Brinch Hansen, "Distributed Process: A Concurrent Programming Concept," CACM, vol. 21, no. 11, pp. 934-941, November 1978.

[4] B. Liskov, "Primitives for Distributed Computing," in Proc. Seventh Symp. on OS Principles, December 1979, pp. 33-42.

[5] A.L. Ambler, et al., "Gypsy: a Language for Specification and Implementation of Verifiable Programs," ACM SIGPLAN Notices, vol. 12, no. 3, pp. 1-10, March 1977.

[6] R.P. Cook, "*MOD - A Language for Distributed Computing," IEEE Transactions on Software Engineering, vol. 6, no. 6, pp. 563-571, Nov. 1980.

[7] J.A. Feldman, "High Level Programming for Distributed Computing," CACM, vol. 22, no. 6, pp. 353-368, June 1979.

[8] G.R. Andrews, "Synchronizing Resources," ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, pp. 405-430, October 1981.

[9] K. Jensen and N. Wirth, Pascal User Manual and Report, Springer-Verlag, 1976.

[10] E.W. Dijkstra, "Guarded Commands, Non-Determinacy and Formal Derivation of Programs," CACM, vol. 18, no. 8, pp. 453-457, August 1975.

[11] J.L.W. Kessels, "An Alternative to Event Queues for Synchronization in Monitors," CACM, vol. 20, no. 7, pp. 500-503, July 1977.

[12] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," CACM, vol. 17, no. 10, pp. 549-557, October 1974.

[13] A. Silberschatz, R.B. Kieburtz, and A.J. Bernstein, "Extending Concurrent Pascal to Allow Dynamic Resource Management," IEEE Transactions on Software Engineering, vol. 3, no. 3, pp. 210-217, May 1977.

[14] H.A. Schmid, "On the Efficient Implementation of Conditional Critical Regions and the Construction of Monitors," Acta Informatica, vol. 6, pp. 227-279, 1976.

[15] D.S. Hirshberg, "Fast Parallel Sorting Algorithms," CACM, vol. 21, no. 8, pp. 657-661, August 1978.