ConC: A Language for Concurrent Programming

Vijay K. Garg
Department of Electrical and Computer Engineering,
University of Texas, Austin
Austin, TX 78712

C. V. Ramamoorthy
Computer Science Division,
University of California, Berkeley
Berkeley, CA 94720

February 22, 1993

Abstract

Present concurrent languages do not support any form of analysis of the communication structure of programs. To support high level specification and analysis of distributed systems, we propose two new constructs- *handshake* and *unit*. The handshake construct is a remote procedure call generalized for multiple parties. The unit construct restricts the possible calls to various handshake procedures, and thereby provides a synchronization mechanism. These constructs are part of a formal model called the Decomposed Petri Net (DPN), which lends itself to automatic analysis. The current system called ConC(Concurrent C) extends "C" for concurrent programming and runs on a Sun cluster under Unix 4.2 BSD.

Keywords: Concurrent Languages, Petri Nets, Multiple-Party Interaction

1 Introduction

Concurrent programs are difficult to design and the simplest of them can have subtle errors. These errors arise due to presence of concurrency and generally result in

violation of a safety or a liveness property. To detect these errors, the system should provide automatic analysis of the communication aspects of a program [26]. For example, it should support queries such as - "Is sequence of events <x,y> possible?", and "Is state S reachable?". Such analysis is difficult in conventional concurrent programming language systems because the synchronization aspects of a program are interwoven with the computational aspects. Also, communication is expressed using low level primitives making it harder to write and debug programs. Therefore, we had two goals in designing the communication primitives - *high level specification* and *analyzability*.

In this paper, we propose two new constructs for concurrent programming - *handshake*, a generalization of the remote procedure call, and *unit*, a communication structuring mechanism. We also assume that the language supports guard construct similar to one provided by CSP. A handshake is shared among two or more processes with one of them acting as its *master*. Each process has a procedure-like interface with a handshake. When all the participating processes call their handshake procedures, the shared handshake body is executed by the master. The unit construct is used to restrict the sequence of possible calls to various handshake procedures and thereby provide a synchronization mechanism between multiple processes. Thus, a unit can be viewed as an automaton that specifies all possible sequences of handshake procedures. The handshake and unit constructs form part of a formal model called the Decomposed Petri Net (DPN) Model[12, 15]. The DPN is theoretically equivalent to Petri Nets and all the analysis techniques for Petri nets such as coverability tree[17] and matrix equations[20] are directly applicable to the DPN.

In our paradigm, we support separation of concerns by separating *computation* objects and *control* objects. Computation objects are specified in any standard sequential programming language such as Pascal, C or sequential Ada. They are used mainly to capture the computation aspects of the system and do not concern themselves with synchronization. Control objects, on the other hand, are written as units. They specify the computation that is directly related to communication. For example, synchronization is handled by these objects. They are mechanically analyzable for most interesting properties as their expressive power is less than Turing machines, (equivalent to Petri nets).

The rest of the paper is organized as follows. Section 2 discusses the related work and distinguishing features of our work. Section 3 discusses the constructs. Section 4 discusses the implementation of these constructs. Section 5 presents the underlying formal model behind units. Section 6 discusses the status of the ConC project and future research directions.

2 Related Work

Andrews and Schneider[2] classify concurrent languages into three categories. The *shared memory based* programming languages assume that variables can be accessed by any process. To guarantee mutual exclusion, constructs such as critical regions and monitors are used. Example of such languages are Concurrent Pascal, Mesa[19] and Modula. The *message based* programming languages provide send and receive constructs for communication. Examples of such languages are CSP[16] and PLITS[9]. The *operation based* languages combine aspects of the other two classes. They provide remote procedure call as the primary means of process interaction. Ada, Distributed Processes[3] and SR[1] fall in this class. Since the handshake extends the remote procedure call for multi-party interaction, it belongs to this class as well. The features that distinguishes the ConC from related efforts are as follows:
(1) **Synchronous Communication**: We believe that users of the distributed systems should not have to deal with asynchronous communication as it makes the program difficult to debug, prove and analyze. In this respect, we agree with the philosophy of programming languages such as Ada and CSP, and differ from PLITS.
(2) **Multi-Process Interaction**: Many applications require interaction between more than two processes and the user can program at high level if such a facility is directly provided by the language. CIRCAL[18], Raddle[10], Multi-way Rendezvous[8], PPSA[24], and Script[11] have also suggested multi-party interaction in one form or another. CIRCAL, Raddle and PPSA allow synchronization based on matching of event names but do not provide a remote procedure call like interface. Script shows how details of multi-process interaction can be hidden but does not provide direct support for the multi-party interaction. None of them support any form of analysis.
(3) **Analysis of Interaction**: As most errors in concurrent systems arise due to erroneous specification of process interaction, any analysis of the interaction will greatly increase the programmer's productivity. None of the above mentioned languages support the analysis. Such analysis is more common for communication protocols which is done mainly for specifications expressed in State Machines, Petri nets or bounded variable programming languages[25]. One of the early attempts to incorporate such analysis in a full fledged programming language was Path Expressions[5]. Path Pascal[6] based on Path expression is, however, a shared memory based language. Also the analysis provided by Path Pascal is not as extensive as that provided by ConC.
(4) **Communication Abstraction Mechanism**: Researchers in programming languages have found abstractions a useful mechanism to increase the understandability of the software. Consequently, current programming languages provide control abstraction through loop constructs and procedure calls, and data abstraction through abstract data types. One of the main functions of an abstraction is to provide only structured access to the primitives. For example a control abstraction mechanism seeks to provide a structured use of goto's. Similarly, the complexity of concurrent software has made it necessary that goto's of the communication world (send, re-

ceive, remote procedure calls etc.) be allowed only in a structured manner. Path expressions specify the sequence of procedures that can be made on shared variables and therefore can be termed as the first attempt for providing such a mechanism. Francez and Hailpern[11] were first to coin the term and use it in their proposal of Script. ConC provides structuring of the communication primitives through the unit construct.

Table 1 summarizes some of the well known concepts that can be shown to be special cases of constructs provided in the ConC.

Feature	**Example**	**ConC**
Synchronous communication	CSP	handshake
Remote procedure call	Ada	parametrized handshake
Multi-process interaction	Raddle	multi-process handshake
Abstraction Mechanism	Script	unit
Path Constraints	Path Pascal	unit expressions
Reachability	Petri Nets	DPN

Table 1: Special Cases of Handshake and Unit Constructs

3 Constructs

3.1 Handshake construct

The remote procedure call has become one of the most favored communication primitive because of its similarity to the local procedure call, a well understood concept. A handshake is a remote procedure call generalized for multiple parties.

A handshake consists of the declaration of handshake procedures and a shared body. The body is executed by the master only when all handshake procedures have been called by their respective processes. Thus, handshake can be used as a synchronization point of multiple processes. For illustration, consider the distributed players problem. Assume that there are four players who are interested in playing various games as shown in Figure 1. Joe is willing to play chess, bridge or poker. Mary is willing to play any of the games while Jack and Bob play only bridge or poker. Playing a game requires *rendezvous* between two or more processes. This is achieved by handshake construct as follows:

The above example illustrated the use of the handshake construct for synchronization. The handshake construct is also useful for communicating data from one process to the other. The handshake procedures may be called with parameters. When the handshake is executed by the master of the handshake, all the parameters are considered available. The body of the handshake can use any of the parameters or its own local variable. As an example, consider the synchronous *send* provided in the Unix as a library facility. The handshake description of such a primitive in ConC is shown in Figure 2. It specifies that when process P1 calls *send* and P2 calls *receive*

```
handshake bridge;
    procedure Joe.bridge();
    procedure Jack.bridge();
    procedure Bob.bridge();
    procedure Mary.bridge();
begin
end ;
```

Figure 1: Distributed Player Problem

```
handshake syncsend;
   const
        MAXLENG = 50;
   type
        message = array[1..MAXLENG] of char;
        numbytes = 0..MAXLENG;
   procedure P1.send( senddata: message; scount: numbytes);
   procedure P2.receive( var recdata: message; var rcount: numbytes);
   var i: integer;
   begin
        for i:=1 to scount do
        recdata[i] := senddata[i];
        rcount := scount;
   end;
```

Figure 2: An Example of a Handshake Specification

with their parameters, the associated body with the handshake is executed by the first process named in the handshake (P1). Note that the syntax is symmetric for caller and callee in contrast to rendezvous in Ada where the callee uses accept and the caller uses entry procedure call to make a rendezvous. Also note that the syntax requires every participant in the process to be explicitly named. We are also assuming a static structure for processes. These restriction are required for the feasibility of automatic analysis of the communication structure.

We next describe the syntax for the handshake construct using BNF. We use {} to denote zero or more repetitions of the enclosed expression.

;a handshake specification
<*handshake-dcl*> ::= **handshake** id **';'** < *global-dcl*>
{<*proc-specs*>} <*local-dcl*> <*body*> ';'
; this section specifies types used in declaring parameters
; it also specifies variables that are owned by the handshake
<*global-dcl*> ::= the usual const, type and var declarations
; headers for various procedures which share the body
<*proc-specs*> ::= **procedure** processid '.' procname ({ <*param*> }) ';'
| **inform** processid ';'
<*param*> ::= [**var**] id ':' <*type*> ';'
<*local-dcl*> ::= local variable declaration
; the body executed by the master of the handshake
<*body*> := the usual programming language body

```
handshake put-item;
      procedure sender.send(sdata: integer);
      procedure buffer.insert(var bdata: integer);
begin
      bdata := sdata;
end;

handshake get-item;
      procedure buffer.remove(bdata: integer);
      procedure receiver.receive(var rdata: integer);
begin
      rdata := bdata;
end;
```

Figure 3: Handshakes for Buffered Sends

Now assume that we have a buffer process available that can store a single message. For simplicity, we assume that the processes are interested in communicating integers only. The handshake declaration is shown in Figure 3.

Sometimes, a process may not want to participate in a handshake in the sense of sending parameters or receiving results. It may simply be interested in knowing if the handshake took place so that it can control some of its other handshakes. The *inform* clause can be used for these situations and its use will be illustrated in a later example.

It may seem that the handshake construct does not promote concurrency as the body of the handshake is executed by a single process. An alternative construct may have separate body for each procedure in the handshake in which they can use input parameters of other procedures. There are two reasons for not choosing this option. Firstly, for abstraction purposes, we provided the notion of permanent data with a handshake. The user would have to worry about consistency of this data if concurrency was allowed within a handshake. Second, the implementation of distributed handshake would lead to many more messages than required for the centralized execution.

The handshake construct is more suitable for message-based systems. In these systems, it is more natural to assume that every object has its caretaker who can access and update its state. The caretaker of a object is modeled as the master of the handshake in our system.

3.2 Unit Specification

In the example of distributed players, players may have different constraints on their sequence of games. For example, Joe may wish to play only tennis after chess. Similarly, in the example of buffered send, we did not specify the buffer process. If the buffer process allowed *put-item* and *get-item* in any order, the communication may be faulty. The buffer behaves correctly if it satisfies the constraint that a *put-item* is always followed by a *get-item* and *vice-versa*. As a result, the sender may have to wait for the receiver to read the item before it sends another item to the buffer process. To express such constraints and therefore provide a high level synchronization mechanism, we provide the *unit construct*.

To describe all possible sequences of the handshake procedures, we can use a algebra based model (e.g. regular expressions) or transition based model (e.g. finite state machines). The unit construct is a transition oriented model. In Appendix 1, we describe an algebra based model called concurrent regular expressions. [15] describes the conversion from one form of specification to the other. It can also be shown that path expressions and COSY expressions are special cases of concurrent regular expressions. The interested reader is referred to [15]. In this paper, we will restrict the discussion to the transition oriented model.

A unit is a directed graph where vertices are called places, and edges between them are labeled by names of handshakes. In addition, there is a concept of tokens which may be thought of as residing in places. A handshake can take place only if there is a token in the tail vertex (source place) of the handshake. After execution, the token moves to the head vertex (destination place). Figure 4 shows the linguistic and graphical equivalent of a one-frame buffer. The marking construct is used to describe the number of tokens at various places. The body of a unit consists of enumeration of all transitions in the unit. These transitions are arranged on the basis of their source places. A place name, such as avail in the above example, is followed by the description of transitions, each consisting of a handshake name followed by the destination place.

A unit also has the notion of an infinite number of tokens which are represented by putting a * in the place. Any number of tokens can be added or removed from a *-place without changing it. Figure 5 presents the unbounded buffer problem in ConC.

Figure 6 shows the linguistic and graphical equivalent of the constraints imposed by Joe. The BNF for the specification of a unit is as follows:

<*unit-specs*> ::= **unit** id ';' <*marking*> **begin** { <*transitions*> } **end** ';'
<*marking*> ::= **marking** { '[' <*placename*> ':' <*num*> ']' } ';'
<*num*> ::= '*' | integer
<*transitions*> ::= placename { > transname placename ; }

```
(* put-item should be followed by a get-item
unit buffercomm;
     marking [unavail:1];
begin
     unavail
          > put-item avail;
     avail
          > get-item avail;
end;
```

Figure 4: Unit Specification of a One-frame buffer

```
(* the receiver must wait for the sender *)
unit buffercomm;
marking [unavail:*];
begin
unavail
     > put-item avail;
avail
     > get-item unavail;
end;
```

Figure 5: An Example of the Unit Specification

```
unit Joecomm ;
marking [bstate:1];
begin
bstate
      > chess cstate;
      > tennis tstate;
      > bridge bstate;
cstate
      > tennis tstate;
tstate
      > bridge bstate;
end;
```

Figure 6: Unit Specification for Joe

As another example of these constructs, consider the mutual exclusion between two processes X and Y. The entire system has four handshakes - *p1in, p1out, p2in, p2out*. *p1in* handshake requires permission from both the processes X and Y but does not require any parameters from process Y. In fact, the computation object of process Y need not even call *p1in*. Therefore, we use inform clause for process Y. This is specified in the handshake declaration of *p1in*. *p1out*, on the other hand, does not need any coordination from the process Y. The unit construct allows *p2in* to happen only if the process X is in a non-critical state. The entire specification of the process X is given in Figure 7.

As another example consider the problem of dining philosophers. This problem, first posed by Dijkstra, requires an algorithm for philosophers who are sitting on a circular table. They are five in number and there is a fork between every two of them. There is a bowl of spaghetti in the center which can be eaten by any philosopher but its tangled nature requires that he use both his left and right forks. A deadlock free solution expressed in the ConC constructs is shown in Figure 8. $get_{i,i+1}$ represents that i^{th} philosopher has taken possession of $i + 1^{th}$ fork. The $philosopher_i$ does not seek possession of $i + 1^{th}$ fork unless he also possesses i^{th} fork. Note the simplicity of the solution due to the availability of synchronous communication. The simplicity comes because the complexity of implementing synchronous communication is buried in the handshake construct. Note that if all philosophers express interest in eating by calling the handshake $get_{i,i+1}$, the underlying implementation may chose any one of them.

Having stated the solution to dining philosophers problem, we would like to verify that our solution is indeed deadlock free. Current programming systems typically require manual analysis for such questions. It is impossible to develop a programming system that proves the correctness of a general program (because most interesting properties such as termination are undecidable). Thus, proof systems for various programming languages have been developed that facilitate manual proofs of assertions on the program. On the other hand, models such as finite state machines and Petri nets (or DPN) may be mechanically analyzable for properties (such as reachability) but do not capture all possible programs. The unit construct supports a paradigm in which an object has two parts: Turing-equivalent computation object and Petri-net equivalent control object.

The unit construct may seem redundant to some readers. There is no analogue of this construct in most conventional languages such as Ada. In fact, any program that can be written using non-trivial unit construct can be written without the notion of unit (i.e. allowing all possible handshakes if participants are ready). However, the program without unit construct have synchronization aspects of a function intermixed with computational aspects. For example, the mutual exclusion using just rendezvous construct (handshake in our case) would be more difficult to program. The unit construct attempts to separate concerns of computation from synchronization. For example, the unit construct in mutual exclusion ensures that both *p1in* and *p2in* are

```
handshake p1in;
    procedure X.p1in();
    inform Y;
begin
end;
handshake p1out;
    procedure X.p1out();
begin
end;

(* communication unit for process X *)
unit mutex1;
marking[noncritical:1];
noncritical ;
    > p1in critical;
    > p2in noncritical;
critical ;
    > p1out noncritical ;
end;

(* internal computation for process X *)
main()
{
    int i;
    (* note that, p2in and p2out do not appear in this object *)
    for (i=1; i<=10; i++)
    {
        p1in();
        (* this is the critical region *)
        p1out();
    }
}
```

Figure 7: Mutual Exclusion Between Two Processes

```
handshake get_{i,i+1}
        procedure philosopher_i.get_{i,i+1};
        inform philosopher_{i+1} ;
  begin
  end;

unit philunit_i;
  marking[neutral:1];
  neutral
        > get_{i,i+1} eating ;
        > get_{i-1,i} waiting;
  eating
        > put_{i,i+1} neutral;
  waiting
        > put_{i-1,i} neutral;
end ;

process philosopher_i;
begin
  if hungry then begin
        get_{i,i+1}();
        eat();
        put_{i,i+1}();
  end;
end;
```

Figure 8: A Solution of Dining Philosophers Problem

not enabled at the same time.

This separation simplifies the task of programming as the programmer can focus initially on computational aspects without worrying about the sequences of handshakes that should not be allowed. For example, in case of distributed player problem he can postpone the task of checking whether the players play game in the right order. The separation of control may help in making the computation object simpler to prove. This is in agreement of the philosophy of Unity[7]. If some safety assertion can be shown to be true for trivial unit (a unit that allows all handshakes) then it is also going to be true when some sequences are restricted. Since handshakes may involve a single process (such as p1out in mutex algorithm), the unit construct is also useful for sequential programming.

Alternatively, the facility of unit construct can be viewed as a meta-level control which ensures that the control in the computation object does not violate any global constraints.

3.3 Guard Construct

For selective communication, we also assume that the language has the guarded command construct as proposed by Hoare for CSP. A guarded command consists of one or more <guard, action> pairs. A guard consists of a boolean condition and optionally a handshake. The handshake is enabled only if the boolean condition is true. If an enabled handshake can be executed (participating processes are willing to execute the handshake), the guard is considered true and the statement corresponding to the guard can be executed. The syntax of the guard construct is as follows:

< *guarded-command* > ::= '[' <*guard*> '->' <*statement* > ']'
<*guard*>:: <*boolean-condition*> '&' handshakeid

For an example of guard construct, consider the buffer process which may communicate with either the sender or the receiver. Its specification is shown in Figure 9.

4 Implementation

Each logical process is actually composed of two real processes: computation and control process. The computation process interacts with control process by two means: (1) Simple handshake call: As seen earlier the execution of a handshake may require the participation of multiple processes. The computation process sends an *enable* message to the control process whenever it is ready for a particular handshake and waits for a reply from it. The control process goes through a series of protocol mes-

```
process buffer;
    int findex = 0;
    int bindex = 1;
    itemtype buffarray[SIZE];
        [
        put-item -> insert(item);
            findex = (findex + 1) mod SIZE;
            buffarray[findex] = item;
        get-item -> remove(buffarray[bindex]);
            bindex = (bindex + 1) mod SIZE;
        ]
end
```

Figure 9: Specification of the Buffer Process

sages with other control processes to agree on the execution of the handshake. If it succeeds, it tells the computation process to proceed and send the relevant message to the master of the handshake. If the handshake is not possible because one of the participant processes has terminated then the control process sends an error message to the computation process.
(2) Calls from Guard: Since only one handshake is allowed in every guarded statement, it can be always be executed if all participant processes are ready for it. The computation process enables all the handshakes that are called from the conditions of the guarded statements. It then waits for a reply from the control process. The control process sends to the computation process, the name of the handshake it has committed. It is the responsibility of the computation process to execute the handshake.

4.1 Handshake Construct

For each handshake the first process named is assumed to be its master. The master is responsible for receiving the value parameters from other processes, executing the code associated with the handshake and shipping back the results. Slave processes send their parameters and wait for the results or continue execution if they do not expect any results back. Our prototype assumes that messages do not get lost, duplicated or corrupted. (It is the job of communication subsystem to guarantee that.)

4.2 Unit Construct

It is the responsibility of control object to ensure that handshakes happen only in the sequences permitted by the unit construct. The control object, however, can execute only those events that are enabled by the computation object. If a process was named in the inform clause of handshake the control object can always execute that handshake. For example, X is just in inform clause of *p2in* and therefore the unit for process X assumes that *p2in* is always enabled. Otherwise, a handshakes may be enabled by a handshake call, or from a guard construct. For example, *p1in* can be executed by the unit process only if X makes a call to *p1in*. An example, where multiple handshakes may be enabled in computation object is the buffer process in which both *put-item* and *get-item* are enabled on execution of the guard.

A unit can execute the event if it is enabled by its computation process and its own structure (i.e. there exists a place with one or more token which has an outgoing edge labeled with that handshake). By our definition of handshake all units that have the handshake must agree on its execution. To arrive at this agreement, the unit processes go through a protocol described in [15]. The protocol ensures that if some handshake is executable then the system will not deadlock. It involves inviting processes to commit for a handshake and then committing it if all processes agree.

4.3 Guard Construct

As the computation process reaches a guard construct, it evaluates boolean condition for all guards. If a guard with just boolean condition is true, then the statement may be executed, otherwise it sends *enable* messages for all handshakes named in guards with their boolean condition true. On receiving the name of the handshake that was executed by the unit process, the computation process can execute the corresponding handshake and guard.

5 DPN: The Underlying Model

To define a decomposed Petri net, we partition it into multiple *units* which share its transitions. Each unit contains some of the places of the original Petri net. Intuitively, the decomposition is such that the tokens within a unit need to synchronize only with tokens in other units. Formally, a DPN (Decomposed Petri Net) D is a tuple (Σ, U) where

- Σ = a finite set of symbols called *transition alphabet*
- U = set of units $(U_1, U_2..U_n)$ where each unit is a five tuple i.e. $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ where:
  - P_i is a finite set of *places*

- C_i is an initial *configuration* which is a function from the set of places to nonnegative integers N and a special symbol '*'. i.e.,$C_i : P_i -> (N \bigcup \{*\})$. The symbol '*' represents an unbounded number of tokens. A place which has * tokens is called a **-place*.
- Σ_i is a finite set of *transition* labels s.t. $\Sigma_i \subseteq \Sigma$.
- δ_i is a relation between $P_i \times \Sigma_i$ and P_i, i.e., $\delta_i \subseteq (P_i \times \Sigma_i) \times P_i$. δ_i represents all transition arcs in the unit.
- F_i is a set of final places, $F_i \subseteq P_i$.

The configuration of a DPN can change when a transition is fired. A transition with label a is said to be *enabled* if for all units $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ such that $a \in \Sigma_i$ there exists a transition (p_k, a, p_l) with $C_i(p_k) \geq 1$. Informally, a transition a is enabled if all the units that have a transition labeled a, have at least one place with non-zero tokens and an outgoing edge labeled a. Thus in Figure 10, *get-item* is enabled only if both p_4 and p_5 have tokens. A transition may *fire* if it is enabled. The firing will result in a new marking C_i' for all participating units, and is defined by
$C_i'(p_k) = C_i(p_k) - 1$
$C_i'(p_l) = C_i(p_l) + 1$.
A *-place remains the same after addition or deletion of tokens.

As an example of a DPN machine, consider the producer consumer problem. The DPN representation for this problem is shown in Figure 10. The producer produces items which are kept in a buffer. The consumer takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. The consumer can execute *get-item* only if there is a token in the place p_4. Note how the *-place is used to represent an unbounded number of tokens. Formally, $M = (\Sigma, U)$ where $\Sigma = \{$ *produce, put-item, get-item, consume* $\}$, $U = (U_1, U_2, U_3)$ where

- $U_1 = (P_1, \Sigma_1, C_1, \delta_1, F_1)$, $U_2 = (P_2, \Sigma_2, C_2, \delta_2, F_2)$ $U_3 = (P_3, \Sigma_3, C_3, \delta_3, F_3)$
- $P_1 = \{p_1, p_2\}, P_2 = \{p_3, p_4\}, P_3 = \{p_5, p_6\}$
- $\Sigma_1 = \{$ *produce , put-item* $\}$, $\Sigma_2 = \{$ *put-item , get-item* $\}$
- $C_1 = \{(p_1, 1), (p_2, 0)\}$, $C_2 = \{(p_3, *), (p_4, 0)\}$, $C_3 = \{(p_5, 1), (p_6, 0)\}$,
- $\delta_1 = \{(p_1, produce, p_2), (p_2,$ *put-item* $, p_1)\}$
- $\delta_2 = \{(p_3,$ *put-item* $, p_4), (p_4,$ *get-item* $, p_3)\}$
- $\delta_3 = \{(p_5,$ *get-item* $, p_6), (p_6, consume, p_5)\}$
- $F_1 = \{p_1\}, F_2 = \{p_3\}, F_3 = \{p_5\}$

Figure 10: A DPN machine for Producer Consumer Problem

U_1 corresponds to the producer, U_2 corresponds to the buffer and U_3 to the consumer.

The above examples illustrate the modeling power of DPN which can easily be seen to be more than finite state machines and less than Turing machines. [15] shows that DPN has the same theoretical modeling power as that of Petri nets[22] but it enjoys many more useful properties which facilitate specification and analysis of distributed systems.

6 Status and Future Directions

The current ConC system consists of two sub-systems: *ConC translator, and DPN analyzer*. Conc translator generates a set of "C" processes from a ConC program. These processes communicate using the semantics of a synchronous handshake in DPN. The execution of a handshake requires synchronization between multiple processes similar to that required by a generalized CSP alternative command[4]. [15] describes an algorithm for multi-process synchronous communication. ConC Translator is implemented on SUN workstations with 4.2 BSD. DPN analyzer analyzes a given DPN for the following type of queries: Is configuration C1 reachable? Is there any configuration with no exits?(potential deadlocks) It is written in Franzlisp and runs on 4.3 BSD.

The proposal for the ConC is unique in that it combines aspects from diverse languages such as CSP, Ada, SCRIPT, Path Pascal, Raddle and SA. The theory combines aspects from algebraic theory, net theory and formal language theory. Due to the limited experience with such constructs, we have attempted to keep them as simple as possible by retaining only the essential features. Some of the limitations of the current proposal and possible solutions are as follow:

1. The current design uses explicit naming of processes in the spirit of CSP. As for CSP, this may prove restrictive and use of port names may be preferred. We have chosen to keep the initial prototype simple and the future design may include port names.

2. The current process allows only synchronous communication primitives. Asynchronous message passing can be specified using an extra buffer process. We chose to keep synchronous primitives only, as reasoning with asynchronous processes is error prone and cumbersome.

3. The current design also restricts the process structure to be static. This implies that unbounded process activation and recursive process activation is not possible. This restriction is a direct consequence of our aim of keeping the construct analyzable.

4. The current design assumes an error free reliable message service. It also does not address the issues of process failures, reliability, exception handling and security. Similarly specification of priority and issues arising due to fairness concerns are not considered here. The notion of time is also missing in the current design.

7 Conclusions

This paper presents two new constructs, handshake and unit to support distributed computation. The handshake construct is a multi-process generalization of the RPC. The unit construct is used to specify the possible sequences of the handshakes and therefore provide a synchronization mechanism between multiple processes. These constructs unify a large number of concepts such as semaphores, monitors, path expressions, input/output, remote procedure call and communication abstraction. These constructs are based on a formal model called the DPN model which is mechanically analyzable. The analysis can be done with respect to reachable configurations of a DPN machine and the language accepted by it.

8 Acknowledgements

We are thankful to Prof. F. Bastani, Dr. Y. F. Chen, Mr. S. DeNitto, Prof. D. Ferrari, Mr. M.H. Kim, Mr. P. Leong, Mr. A. Prakash, Mr. S. Shekhar, Mr. Y. Shim, Mr. J.S. Song, Mr. J. Srivastava, Prof. C. J. Stone, Mr. W. Tai and Prof. L. Zadeh for their comments on this work. This work was in part supported by RADC under contract 1-482427-26979 and in part by ONR under contract N00014-88-K-0408.

9 References

1. G.R.Andrews, "The Distributed Programming Language SR - Mechanisms, design and implementation", Software Practice and Experience 12, 8 ,Aug 1982, pp 719-754
2. G.R.Andrews, F.B.Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol. 15, No. 1, March 1983, pp 3-43.
3. P. Brinch Hansen, "Distributed Processes: A concurrent Programming Concept", Comm. ACM 21, 11, Nov 1978, pp 934-941.
4. G.N.Buckley, A.Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP", ACM transactions on programming languages and systems (TOPLAS), April 1983.
5. R.H.Campbell, A.N.Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, vol 16, Springer Verlag, New York 1974, pp 89-102.
6. R.H.Campbell, R.B.Kolstad, "Path Expressions in Pascal", Proc. 4th International Conference on Software Engineering, Munich, IEEE New York, 1979, pp 212-219.
7. K.M.Chandy and J. Misra, "Parallel Program Design", Addison-Wesley, 1988.
8. A. Charlesworth, "The Multiway Rendezvous", ACM Trans. on Programming Languages and Systems, Vol 9, No.2, July 1987, pp 350-366.
9. J.A.Feldman, "High Level Programming for Distributed Computing", Comm. ACM 22, 6, June 1979, pp 353-368.
10. I.R.Forman, "On the Design of Large Distributed Systems", Proc. International Conference on Computer Languages, 1986.
11. N.Francez, B.Hailpern, "Script: A Communication Abstraction Mechanism", Proc. of 2nd Symposium on Principles of Distributed Computing, 1983.
12. V.K.Garg, Specification and Analysis of Concurrent Systems Using the STOCS model, Proc. of Computer Networking Symposium, Washington D.C. 1988.
13. V.K.Garg, Analysis of Distributed Systems with many Processes, International Conference on Distributed Computing Systems, 1988.
14. V.K.Garg, C.V.Ramamoorthy, "High Level Communication Primitives for Concurrent Programming", Proc. IEEE International Conference on Computer Languages, Miami, Florida, 1988.
15. V.K.Garg, "Specification and Analysis of Distributed Systems with a Large number of Processes", Ph.D. Dissertation, University of California, Berkeley, 1988.
16. C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.
17. R.Karp, and R.Miller, "Parallel Program Schemata", RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New York (April 1968).
18. G.J.Milne,"CIRCAL and the Representation of Communication, Concurrency and Time," ACM TOPLAS, 7(2), pp 270-298, April 1985.
19. J.G.Mitchell, W.Maybury, R.Sweet, "Mesa Language Manual, version 5.0" Rep.

CSL-79-3, Xerox Palo Alto Research Center, April 1979.
20. T. Murata, "Modeling and Analysis of Concurrent Systems", in book Handbook of Software Engineering, ed. C.R.Vick and C.V.Ramamoorthy, Publ.Van Nostrand Reinhold, pp 39-63, 1984.
21. R.M. Needham,A.J.Herbert, "The Cambridge Distributed Computing System", Publ. Addison-Wesley Publishing Company, 1984.
22. J. Peterson, Petri-Net Theory and Modeling of Systems, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.
23. S.Ramesh, "Programming with Shared Actions: A methodology for developing Distributed Programs", Ph.D. Dissertation, IIT Bombay, India, June, 1986.
24. S.Ramesh, "An Efficient Implementation of CSP with Output Guards", Proc. of International Conference on Distributed Computing, 1987.
25. C.A.Sunshine,"Survey of Protocol Definition and Verification Techniques", Proc. of the Computer Network Protocols Symposium, Liege, Belgium, 1978.
26. R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs", Communications of the ACM, 26(5) pp 362-376, 1983.

10 Appendix 1: Concurrent Regular Expressions

Since regular expressions specify the computation of essentially a sequential finite state machine, they are not very suitable for expressing the languages of the concurrent systems. To specify the trace of a concurrent system, we have proposed an extension of regular expressions (r.e.) called concurrent regular expressions (c.r.e.). Recall that an r.e. over an alphabet Σ is defined as follows:
1) Any a that belongs to Σ is an r.e.
2) If A and B are r.e., then so are A.B (concatenation), A+B (or) and A* (Kleene closure).

For example, consider two sets A and B as follows:
$A = \{ab\}$ and $B = \{ba\}$
then $A + B = \{ab, ba\}, A.B = \{abba\}, A* = \{\epsilon, ab, abab, ababab, ...\}$
$a * b + b * a, abb, ab + ba$ are some other examples of regular expressions. To define concurrent operations, it is especially useful to be able to specify the interleaving of two sequences. With this motivation, we define a operator called interleaving, denoted by $||$. For the sets A and B as defined above $A||B = \{abba, abab, baab, baba\}$

Note that similar to $A||B$, we also get a set $A||A = \{aabb, abab\}$. We denote $A||A$ by $A^{(2)}$. We use parentheses in the power to distinguish it from the traditional use of the power i.e. $A^2 = A.A$. With this notation, it becomes useful to ask if there is an analogue of a Kleene-Closure for interleaving operator. Indeed, it is very useful to

define, what we call $\alpha - Kleene$ closure of a set A, denoted by A^{α} as follows:
$A^{\alpha} = \cup_{i=0} A^{(i)}$

In the above example, $A^{\alpha} = \{w | w \in (a,b)*, \#a's >= \#b's$ for any prefix $, \#a's = \#b's\}$
Note the difference between ordinary closure and the α closure. Also note that the α closure can not be expressed using ordinary r.e. operators.

To provide synchronization, we define a composition operator denoted by []. Intuitively, this operator ensures that all events that belong to two sets occur simultaneously. Or more formally, $A[]B = \{w | w/\Sigma_A \in A, w/\Sigma_B \in B\}$
where w/Σ_A represents the string w restricted to symbols in Σ_A.
In our example, $A[]B = \phi$ as there cannot be any string that satisfies event sequences of both A and B. This corresponds to a deadlock where A waits for B to take action a and B waits for A to take the action b. Consider another set $C = \{ac\}$. Then $A[]C = \{abc, acb\}$. With these additional operators, we define a concurrent regular expressions (c.r.e.) over an alphabet Σ as follows:
1) Any a that belongs to Σ is a regular expression (r.e.). If A and B are r.e., then so are A.B (concatenation), A+B (or), A* (Kleene closure).
2) A regular expression is also a unit expression. If A and B are unit expressions, then so are $A||B$ and A^{α}.
3) if $A_1, A_2..A_n$ are unit expressions then $A_1[]A_2[]..[]A_n$ is a concurrent regular expression.

[Garg 88d] shows that there exists a 1-1 correspondence between the language expressible as the DPN and concurrent regular expressions.