

1121

**ANALYSIS AND DETECTION OF MESSAGE PASSING
DISCREPANCIES IN CONCURRENT PROGRAMS**

DISSERTATION SUBMITTED BY

JISNU GHOSH

**IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR DEGREE OF
MASTER OF TECHNOLOGY**

IN

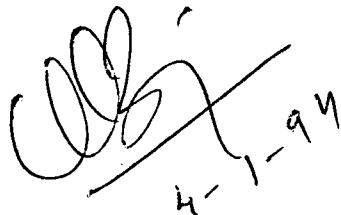
COMPUTER SCIENCE&TECHNOLOGY

**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110 067
JANUARY 1994**

CERTIFICATE

This is to certify that the dissertation entitled "Analysis and detection of message passing discrepancies in concurrent programs" being submitted by me to Jawaharlal Nehru University, New Delhi in the partial fulfilment of the requirements for the award of the degree of **Master of Technology** is a record of original work done by me under the supervision of **Prof.P.C.Saxena**, Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University during the year 1993, Monsoon semester.

The results reported in this dissertation have not been submitted in part or full to any other University or Institute for the award of any degree or diploma etc.



4-1-94

Prof K.K.Bharadwaj
Dean,
School of Computer
and System Sciences,
J.N.U., New Delhi.

Jisnu Ghosh

Jisnu Ghosh



Prof.P.C.Saxena
Professor,
School of Computer
and System Sciences,
J.N.U., New Delhi.

To

my parents

ACKNOWLEDGEMENT

I confer my gratitude to **Prof.P.C.Saxena**, Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi for his precious and generous guidance which has been indispensable for successful completion of the dissertation. I am very much indebted to him for his personal involvement with my work and for providing me with valuable notes and references.

I extend my sincere thanks to **Prof K.K.Bharadwaj**, Dean, School of Computer and System Sciences, Jawaharlal Nehru University for providing me with the environment and all the facilities required for the completion of my dissertation.

I also take this opportunity to thank all faculty and staff members, my friends and well wishers who helped me in every possible way. Specially I bestow my appreciation to Abdaal and Manoj for their continuous encouragement throughout different moments of time.

Jisnu Ghosh
Jisnu Ghosh

CONTENTS

CHAPTER ONE-INTRODUCTION1
1.1 Trends in Computing	
1.2 Relevance Of the project	
1.3 Organization of the report	
CHAPTER TWO-COMMUNICATION IN CONCURRENT PROGRAM8
2.1 Examples of Concurrency	
2.2 Process	
2.3 Types of Concurrency	
2.4 Communication between processes	
CHAPTER THREE-MESSAGE PASSING DISCREPANCY32
3.1 Discrepancy	
3.2 Aim of the project	
CHAPTER FOUR-GLIMPSE OF CONCURRENT C44
4.1 Interprocess communication	
4.2 Process,Process types, Transaction	
4.3 Process bodies	
4.4 Process states and termination	
4.5 Transaction calls	
4.6 An example program	
4.7 Delay statement	
4.8 Timed transaction call	
4.9 Nested process	

CHAPTER FIVE-IMPLEMENTATION57
5.1 Queue Generator	
5.2 Detector	
CHAPTER SIX-CONCLUSION84
BIBLIOGRAPHY88

CHAPTER ONE

INTRODUCTION

1.1 Trends in Computing

During the last few decades a large number of developments have occurred in the field of computer science. The storage capacity of the memories have increased to the point where they equal a significant fraction of human brains storage capacity. The speed and reliability of the systems have been improved dramatically. A large number of impressive software tools have been developed. Increase in device speed, reliability and reduction in hardware cost and physical size have greatly enhanced performance. Given these new hardware and software systems and our improved understanding of the physical world, we are in the threshold of exploring a new horizon of exciting innovations. With the development of more complex systems we are witnessing that the mainstream use of computers are experiencing a trend towards ascending level of sophistication. In the earlier days computers were only used for data processing i.e. for number crunching purposes. But as the accumulated knowledge bases expanded rapidly there is a demand to use computers for knowledge processing. Today's computers can be made very knowledgeable but far from being intelligent. Intelligence is very difficult to create and its processing is even more difficult. Computers are still unable to communicate with human beings in natural forms like speech and written languages, pictures and images, documents and illustrations, computers are far from being satisfactory in

performing theorem proving, logical inferences and creative thinking. We are in a period where computers are used not only for data-information processing but also towards developing practical intelligent systems to advance human civilization. But this requires massive computational power and storage capacity by computers. For this reason high performance computers are increasingly in demand in the areas of artificial intelligence, expert system, industrial automation, remote sensing, weather forecasting and so on. Without superpower computers many of these challenges to advance human civilization cannot be made within a reasonable time period. This can only be achieved by exploring parallelism. Parallelism is a demonstrated success in making programs run faster. It is a conceptual model of tantalizing potential. There are many applications which are computationally intractable for sequential machines. The speed of electric current flow along a conductor is one of the physical phenomena which ensures that such machines can never deliver the performance demanded by the seemingly insatiable user. Parallelism is the answer to satisfy this demand. Parallelism most observers agree can revolutionize computing from supercomputer down to garden variety household workstations. At the workstation level the implications might be even more exciting than they are for supercomputing. Powerful engineering workstations capable of present day supercomputer performance will exploit concurrency not only for raw compute performance but also to provide real time 3-D graphics displays. The ability both to compute and to visual-

ize solutions of complex systems of equations will soon become an indispensable tool of all scientists and engineers.

In programming a scientific or engineering problem a conventional uniprocessor require the problem to be cast into sequential form for its solution. In programming such problem in a multiprocessor computer liberates the programmer from this sequential straightjacket and natural parallelism of the program can be exploited. Most real life science and engineering problems decompose into many subtasks that can be performed concurrently. Only parallel system can acheive a computational throughput which is not acheivable in uniprocessor.

But dealing with parallel computers we have to face a challenge. The challenge is of efficiently programming a parallel machine. The hardware cost is often only a small part of the total cost of solving a problem. Software development on such machines are at present less straightforward than for sequential machines.

A parallel machine consists of many subcomputers that can be focused simultaneously on the same problem. To use a parallel machine, we need programs that do many things at once. The problem is broken down into a number of parallel tasks. Each task is designed to solve a specific portion of the total problem. The successful completion of all the subtasks causes the successful solution of the whole problem. In solving the problem the subtasks can perform independantly of each other or can work in a co-ordinated way depending on

each other on some particular occasion. Though a certain bunch of tasks can be performed simultaneously C, LISP, FORTRAN, BASIC and other conventional sequential languages provide no tools for creating parallel tasks and coordinating their activities. They lack the necessary adjectives and verbs. Programmers need new tools, either new programming languages or new dialects of the old languages or runtime libraries of system level routines in order to write parallel programs. For these reasons concurrent programs are used for programming massively parallel computers. Concurrent programming is becoming increasingly important because multicomputer architectures, particularly networks of processors are rapidly becoming attractive alternatives to traditional maxicomputers. Concurrent programming is important for many reasons;

1. Concurrent programming facilities are notationally convenient and conceptually elegant when used for writing systems in which many events occur concurrently for example in operating system, real time system and database systems.
2. Inherently concurrent algorithms are best expressed when the concurrency is stated explicitly, otherwise the structure of the algorithm may be lost.
3. Efficient use of multiprocessor architectures require concurrent programming.
4. Concurrent programming can reduce program execution time even on uniprocessor by allowing input output operations to run in parallel with computation.

1.2 Relevance of the project

As stated above concurrent program consists of a number of subprograms. Each subprogram can run on different processors. In most cases, during the execution, the subprograms need to communicate with each other. The different parts of the problem, the concurrent program is trying to solve, may be dependant on each other. So, the intermediate results generated in one subprogram can be used by another subprogram for its further execution. For this reason the subprograms are required to transfer data between each other or are required to share some common data. One type of communication is by using synchronised message passing. Here one subprogram sends messages to other programs, or receives messages from another program. In this type of communication when a subprogram tries to send a message to another subprogram it waits till the receiving program accepts that message and then it goes on for further computing. It is also true for a receiving subprogram, this subprogram also waits till it gets a message from an appropriate sending subprogram and then it goes on for its remaining computation. Now concurrent programs consists of a number of subprograms and each may need to communicate with each other. One problem that can appear in developing large concurrent programs is that the message passing between different subprograms may not be done in proper order. There could be mismatch between different subprograms from the

point of message passing. The result is that some subprograms can go in a continuous waiting state. So for successful running of the concurrent program we have to be sure that there is no message passing discrepancies between different subprograms. To check that manually will be a very time consuming and error prone job, specially for large softwares. In the project work a model has been provided which can detect this type of discrepancies in concurrent programs which are using synchronised message passing primitives for interprocess communication. The program is developed to detect message passing errors for programs written in Concurrent C language.

1.3 Organization of the report

In the next chapter i.e. chapter II communication methods in concurrent programs has been described in general. The specific importance is given on the message passing model of communication between diferent processes which is the main area of this project work.

In chapter III the problem of message passing discrepancies between different processes running simultaneously has been discussed thoroughly. Also the importance of the problem from different point of view is shown.

In the project work Concurrent C language has been used as the target language on which the detection scheme will be applied. So, in chapter IV the Concurrent C language

has been described briefly. Mainly those features of the language which is needed for the implementation purpose has been discussed thoroughly.

Chapter V is for implementation. Here the details about the model and the implementation of it for the Concurrent C language is detailed.

Chapter VI is conclusion. Here the achievement and limitation of the project work has been discussed. Further development which can be done in this area is also highlighted.

CHAPTER TWO

COMMUNICATION IN CONCURRENT PROGRAM

Concurrent program can be defined as a program which consists of a number of subparts which are designed to run simultaneously. Concurrency has been present in computers for almost as long as computers themselves existed. Earlier during the development of digital computer it was realised that there was an enormous discrepancy in the speeds of operation of electro-mechanical peripheral devices and the purely electronic central processing unit. The logical resolution of this discrepancy was to allow the peripheral device to operate independantly with the central processor to make productive use of the time that the peripheral device is using, rather than waiting for a slow operation to complete. Concurrent programming as a discipline has been stimulated primarily by two developments. The first is concurrency which has been introduced in the hardware, and concurrent programming could be seen as an attempt to generalise the the notions of tasks being allowed to proceed largely independantly of each other, in order to mimic the relationship between the various hardware components. In particular the control of specific hardware component is often a complex task requiring considerable ingenuity on the part of the programmer to produce a software driver for that component. If a way could be found by which those aspects of the driver which are concerned with the concurrent activity of the device might be separated off

from other parts in the system, the task is eased tremendously. If concurrent programming is employed, then the programmer can concern himself with the sequential aspects of the device driver, and only later must he face the problem of the interactions of the driver with other components within the system. The second development which leads directly to a consideration of the use of concurrent programming is a rationalisation and extension of the desire to provide an operating system which would allow more than one user to make use of a particular computer at a time. The introduction of concurrent programming techniques was also recognised to be a useful tool in providing additional structure to a program.

2.1 Examples of Concurrency

An example of concurrency can be seen by considering the evaluation of an arithmetic expression. Suppose one wishes to evaluate the expression:

$$(a*b+c*d**2)*(g+f*h)$$

It is assumed that the identifiers a,b,c etc have values associated with them and the priority rules for evaluation of the expression are as would be expected. A tree may be drawn showing the interdependencies of the subexpressions within the whole expression and one may use this tree to identify possible concurrency within the evaluation. Three concurrent evaluations of subexpressions can begin at once, namely, $a*b$, $d**2$, and

f*h. When the second and third of these are finished, the multiplication by c and the addition of g can take place respectively, also in parallel. It is only after $c*d**2$ has been evaluated that the sub expression a*b can be added, and then finally the evaluation of the whole expression can be completed.

2.2 Process

Process is a very important notion for concurrent programming. It can be stated that the basic building block for concurrent program is process. One informal definition of process can be stated as that which runs on a processor. But this requires to know what is meant by a processor. For general purpose, processor consists of a device which is capable of accessing other devices in order to retrieve information or send information to that device.

A formal model of process in terms of a set of state variables can be given. At any given moment of time, each of the state variables will contain a particular value and this collection of values is known as the state of the process. The behaviour of the process can be described in terms of an action function which maps from one state to another. The action function is completely determined by the design of the hardware on which the process is running.

2.3 Types of Concurrency

Concurrency in a program can be harnessed in different ways. As for example we can have real concurrency or pseudo concurrency. It is almost always the case that the system allowing the use of multiple concurrent processes will require more processes than the available processors. In those rare cases when the number of process will be lower than the available physical processors we can have real concurrency where each process will run on different processor. In the more usual situation where the program will require more processes than there are processors available, in order not to restrict the system arbitrarily, it is necessary that some mechanism be provided which will simulate the action of a number of processes using single processors only. This may be achieved by running the processor under the control of a program commonly called kernel. Here time division multiplexing is provided. Concurrency provided in this manner is called pseudo-concurrency.

2.3.1 Statement Level Concurrency

The fundamental notion required to specify concurrency is to have some basic constructs which may be regarded as sequential in the ordinary programming language sense. Concurrency can be achieved in instruction level, i.e. the granularity of the parallel activity could be

at the level of individual machine instruction. A slightly more structured view of the concurrency could be taken by considering not machine instruction level, but by considering a single high-level language statement to be the primitive construct.

One of the earliest notations proposed was that of Dijkstra which is **parbegin** and **parend**. Since then a number of authors have used **cobegin** and **coend** for the same meaning. As with the sequential begin and end as found in a language such as Pascal the cobegin and coend are used to bracket a group of statements. Thus we might expect to find that the definition of the language might include the BNF description:

```
<concurrent statement> ::=  
    cobegin<statement list> coend
```

The action of the concurrent statement implies a certain synchronisation of concurrent activity, both when the concurrent statements begin, and when it completes.

2.3.2 Program Level Concurrency

Every operating system which has the facility for providing simultaneous interactive access to multiple users or for processing multiple parallel streams of jobs, must be able to handle a set of concurrent processes. Even if the system has totally static structure in which no additional processes are created and no processes are destroyed during the life of the system, concurrent processing will require some kind of view of what a process is. In the simplest

possible case, there must at least be one process to look after each of the interactive terminals, even if the invocation of a program by a user causes that process to call the program as procedure. Other systems may take the view that the creation of a new process is required whenever a user wishes to start a new program executing. The creator of the new process may or may not be suspended until the new process terminates. In such system, it may be said that the granularity of concurrency is the whole program. Concurrency within UNIX operating system has the granularity of the whole program, and it uses this later technique to execute programs and commands at request of the user, although the user may specify whether the parent process is to regain control immediately or to wait until the child process terminates. User programs may however create new processes for their own purposes, although they may still only execute whole programs within a process.

2.4 Communication Between Processes

As long as all of the concurrent processes are proceeding completely independently of each other, we would expect them all to continue at their own speed until they terminate. If this does not happen, that is if the result of a process are affected by the presence or absence of another supposedly independent process, then we have to investigate the underlying mechanism to find the reason for this problem. For the purposes of the discussion of the concurrent

processes themselves, they will have an effect on each other if they are required to communicate with each other.

If two processes wish to communicate with each other, then this implies that they need to share some common information. If this were not so, then the two processes would be totally independent of each other and could proceed in parallel without any interference between them. Thus some information or resource, is to be shared by two or more processes. Shared resources may be regions of memory or peripheral devices to which both processes require access. Sometimes simultaneous access to a resource by more than one process is permissible, but more frequently it will be necessary to impose the restriction that only a limited number of processes can have access to the resource at any one time.

The communication between concurrent processes can be considered of having two guises, interference and communication. Interference is generally regarded as an occurrence in which one process is able to interfere with the progress, and more importantly with the outcome or results of another process. On the other hand cooperation is regarded to be a generally desirable feature which only affects the behaviour of the participant processes in a constructive way.

Processes executing concurrently and independently may proceed at their own rate and no assumptions may be made about the relative times at which they carry out their

individual actions. Even with pseudo concurrency the primitive operations of each process may be executed at arbitrary moments in time with respect to the times at which other processes' primitive operations are carried out. It will be true of course that the primitive operations within each process will be executed in the correct order, but those operations may be interleaved with the operations of other processes in a completely arbitrary way.

Now some high level constructs will be discussed by which a structured approach to the inter process communication problem can be proposed. In the same way that the use of high level languages allow the programmer to express his algorithms in a more natural way and hence the inclusion of simple logical errors less likely so the use of high level synchronisation constructs in concurrent programs will also tend to reduce the incidence of elementary concurrent programming errors.

2.4.1 Shared Data

In this type of communication between processes the processes wishing to communicate with each other do so not by addressing each other but by accessing data which is known and available to them all. This type of construct also provides way of accessing the shared data which ensure that the data itself is not compromised by undesirable simultaneous accesses by competing processes by offering

operations in the form of procedures and functions to control the manipulation of data. The data structure itself is purely passive, and changes are made to the structure by allowing procedures and functions to be called by the active elements in the system, namely processes. As these procedures are called they in a sense become part of the process which calls them, and the data itself temporarily becomes part of the address space of the calling process. Here the competing processes are not required to have any knowledge of the identities of their competitors, but merely to know the name of the object(s) they wish to access and which operations they wish to apply to the data.

Different types of program structures are used in shared data construct.

Critical Regions

Brinch Hansen presents a program structure, originally proposed by C.A.R. Hoare which guarantees a correct use of critical sections and provides a method of ensuring that shared variables are only accessed within an appropriate critical sections. This construct is called critical region. Brinch Hansen suggests that it should be possible to declare variables to have the attribute shared, and he proposes that an additional control structure called region should be provided. The region statement is used in a similar manner to that in which the Pascal with statement is used but the subject of the region statement is required to have the

attribute shared. The semantics of the region statement then require that exclusive control of this shared variable is necessary before the body of the region is executed.

The following program gives an example of the use of shared data construct;

```
type D=...;
var v:shared D;

begin
  initialise(v);
cobegin

  P1: repeat
    region v do critical_section_1;
    non_critical_section_1
  until false;

  P2: repeat
    region v do critical_section_2;
    non_critical_section_2;
  until false;
coend
end
```

The advantage of this type of construct is that the compiler handling construct of this type will ensure that the shared resource (the shared variable v in this case) is only accessed within the respective critical sections.

Monitor

Though the critical region construct has the ability to check that the shared data is not accessed outside the critical region, however, they do not apply any discipline to the way in which the shared data is manipulated. That is once approval has been obtained for access to the shared data,

there are no constraints on the operations which maybe performed on the data. There is, therefore , an obligation on the programmer to ensure that the manipulations carried out on the data structure during the execution of the critical region donot leave the structure in an inconsistent state,i.e. the invarient is true on exit from the critical region.

To impose some controls over the way in which a shared data structure is manipulated monitor type of synchronisation construct is used. This is an extension of the notion of class as found in some programming languages.

The class allows the programmer to manipulate a data structure in a controlled way by making access to the data impossible except through a defined interface, consisting of a set of functions and procedures. Thus it can ensure that unconstrained interface with the data is impossible, since the creator of the class can define those and only those operations on the data which do not destroy the consistency of the data structure. Another feature of the class construct is that it encourages the use of abstract data types upon which operations can be defined without the client of these operations being burdened with the implementation details of the data type. By compelling the user of the data structure to access it only through the defined operations, the author of a class can ensure that the concrete representation of the data(i.e. the local variables of the class)is in a consistent

state on exit from the class. Furthermore the principle of the class can be used to group together data and operations in a single structure. This property of the class is precisely what is required to maintain the integrity of a data structure when it is desired to operate upon it with a number of concurrent processes.

The monitor construct, proposed by Hoare, provides the notion of shared class, but goes on to insist that any process wishing to execute an operation of the class may do so only if no other process is currently accessing the data. So, the monitor can be considered as a set of hidden variables together with a set of visible procedures and functions, but with additional restriction that only one process may be executing any of the procedures at a time. Monitor is also like shared variables because access to the components of the shared variable is only permitted using particular sections of the code. The principle difference is that the monitor provides a single object in which the data and the operations on that data are collected together in one place within the program. In other words it is not necessary to broadcast all the details of the data structure to all the processes which might wish to use it, but merely provide operations which will manipulate the data structure on behalf of the processes.

As with the critical region, so with the monitor, there is logical problem associated with giving exclusive rights of access to a data structure to a process which may be unable to use it for other reasons. It is then necessary for this process to relinquish control in order to let a second process have access to the data, while at the same time keeping some rights of access so that operations may be performed when the inhibiting condition is removed. The monitor, therefore, like critical region, requires a mechanism for allowing a process which has control of the monitor, to relinquish temporarily in order to allow another process to make modifications to the data structure and thus allow the first process to continue. Such mechanism is provided within the monitor and is called a condition. A condition is similar to semaphore and actually the condition is also manipulated by using two primitives signal and wait which behave in similar manner to the corresponding semaphore operations. A variable of type condition may only be declared within the monitor however, and therefore the operations can only be invoked by a process already in a monitor procedure or function. The effect of calling a wait operation on a condition is to cause the calling process to be suspended and to give up temporarily control of the monitor. Another process may then enter the monitor, and it is expected that some other process will eventually invoke the signal operation on the condition, at which time the waiting process can be resumed.

A simple example program with monitor construct is shown below to illustrate the idea;

```
monitor
  BEGIN
    var busy:Boolean;
        Nonbusy:condition;

    procedure acquire;
      begin
        if busy then Nonbusy.wait;
            busy=true;
        end;
    procedure release;
      begin
        busy=false;
        Nonbusy.signal;
      end;
    busy=false;
  END
```

Here we have a single resource for which a number of processes are competing and the characteristics of the process is that only one process should be able to use the resource at any one time. It is therefore necessary to impose some constraints on the access of the resource. The two variables busy and Nonbusy used here, not known outside the monitor, restricts the access of the resource.

Though the behaviour of a condition is similar to the semaphore but there are two very significant differences between them. Firstly the wait operation on a condition variable will always cause the calling process to suspend itself, unlike the semaphore wait which will decrement the semaphore counter and then only wait if the resulting value is negative. Thus the behaviour of a program using a semaphore will depend crucially upon the initial value given

to the semaphore counter, a condition variable requires no initialisation. The monitor conditions are generally associated with boolean expression which is generally tested before the wait is called, and this test in some sense replaces the test of the counter value during the semaphore wait operation.

The second difference arises from the fact that a signal on a condition variable has no effect if no processes are waiting.

2.4.2 Message Passing

It is another technique for communication between concurrent processes. In this method each process has its own self contained state space or address space which is not shared by any other processes either as a whole or in part. Here the message passing operations are primitives which are part of the underlying architecture providing concurrency support and are available to any process wishing to use them.

The simplest form of interaction between two processes P1 and P2 is for one of them (P1) to send a message to another (P2). For this P2 must call the primitive receive and P1 must call send, specifying both the message to be sent and the intended recipient(P2). So, it is necessary for P1 to be aware of P2's identity. Here in this example it is not necessary for P2 to be aware of the identity of P1, not

even after the message has been delivered. Two operations required can be represented like shown below,

```
and      procedure send(p:ProcessId;m:MessageFormat)
          procedure_receive(var m:MessageFormat)
```

Here in this case the reliability of the message transfer mechanism is not considered. For the transfer to occur actually, there must be a degree of synchronisation between the communicating processes. Synchronisation depends on the implementor of the underlying system, particularly as the individual processes are supposedly unaware of the passage of time if for any reason they are unable to make progress. Thus the semantics of the operation send are that the procedure is complete when the message has ceased to be the responsibility of the sending process. If the process should happen to be delayed for any reason while attempting to send a message, the process will not be aware of the delay. Similarly, the process wishing to receive a message will be delayed until a message is available. Having called the receive primitive, the process will not be aware of any further progress until the receive completes by delivering the incoming message. The sending and receiving process may be required to synchronise absolutely in order to transfer a message, meaning a send must wait until a matching receive is invoked, after which the message transfer takes place, and

the two processes can continue along their respective execution paths. Alternatively the underlying system may be capable of buffering messages, in which case an attempted send will complete as soon as the buffer mechanism has taken the message and placed it in its own private memory space ready for delivery when an appropriate receive call is made. Since the sending and receiving processes are unaware of any delays there may be within the message transfer mechanism, distinctions such as these are of no interest to the communicating processes. But there are variations of message transfer mechanism which are of interest to and may have consequences for the users of send and receive primitives.

In the first simple example stated on message passing mechanism the receiving process had no knowledge of the source of the received message but for most practical cases the receiving process needs to know the source of the received message. A possible example is when a process is offering a service of which a number of processes (which are called clients) may want to avail. So, the process offering the service , which is the server, will need to carry out an action on behalf of a client , and then reply to the client indicating the results of the action. In such cases it is necessary for the server to know the identity of the client process, since it could be any of the clients which requested

the service. A modification to the previous receive operation stated will make this possible as shown below;

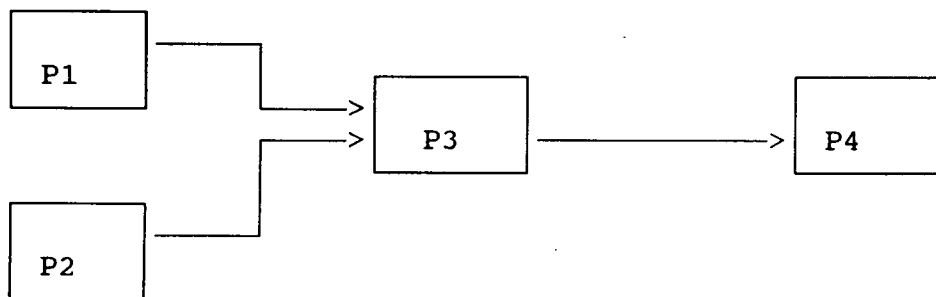
```
procedure receive(var p:ProcessId;var m:MessageFormat)
```

The two procedures send and receive as defined represent the simplest possible mechanism for passing messages between processes. Any synchronisation which may take place as a result of passing messages is not significant as far as the component processes are concerned. There are however some possible alterations which may be made to the message system which do affect the behaviour of the communicating processes.

In the procedure receive the calling process is given the first message which was sent to that process. If for any reason the process wishes to receive a message from a specified sender then the receiving process must take responsibility for accepting all of the messages sent to it, and dealing with them at a later time. This responsibility could be in the form of simply replying to the sender of each unwanted message, asking for the message to be sent at a later time. When the awaited message arrives the receiver can take the required action and then return to the problem of the messages which arrived in the interim or waiting for resubmission.

It is possible for the message system to handle it on behalf of the processes, however, by allowing the user of the receive primitives the option of specifying the process from which a message is to be received. Clearly it would not be desirable for this to be the only way of receiving the messages, but in some instances it would be more convenient for the system to handle the queuing of unwanted messages rather than placing this responsibility with the user process.

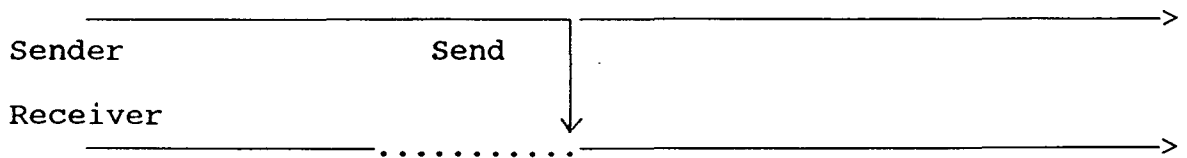
In the previous paragraph the concept of client server relationship has been introduced, now we consider a situation like that shown in the figure below



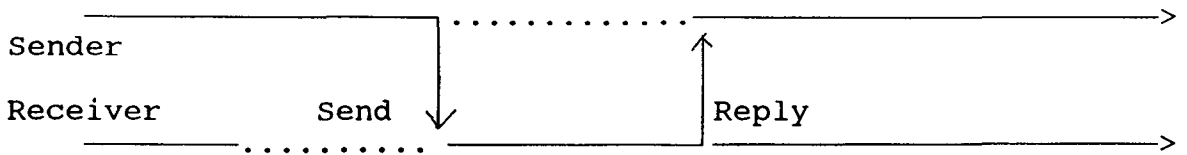
Process P3 is a server which offers services, and say P1 and P2 are two clients of P3. It may be frequently the case that the server process while serving a request from a server

(say P1) may discover that it itself needs a service from another server say P4. The process P3 can therefore act both as a client and a server at the same time. Let us take the case that while processing a message from P1, P3 sends a request to P4. P3 will be unable to complete the service for P1 until it is receiving the service from P4 and it therefore will have to call the receive procedure to accept the message returned from P4. Now one thing can happen, if P3 simply accept the next message sent to it, then the message can come from any of the other processes such as from P2. So, P3 has to remember the message from P2 and process it at a later time or else send a reply to P2 asking it to send the message at a later time. Selective receive are used for this type of situation. Here the receiving process specifies the process from which it wants to accept the next message. A possible application is where a server wants to restrict his client processes for safety or security purpose of the whole system. A special case of the selective receive is to extend the message system to include a reply primitive.

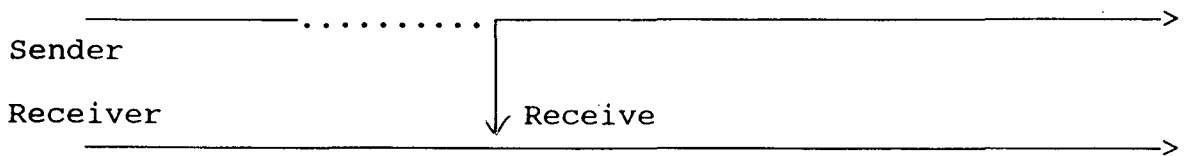
This operation looks like send but is only used following a receive to acknowledge that the message has been received. The send itself is modified so that the sending process is blocked until the reply is received. The unconstrained send and receive and send with blocking until the reply is received is shown in the adjoining figure.



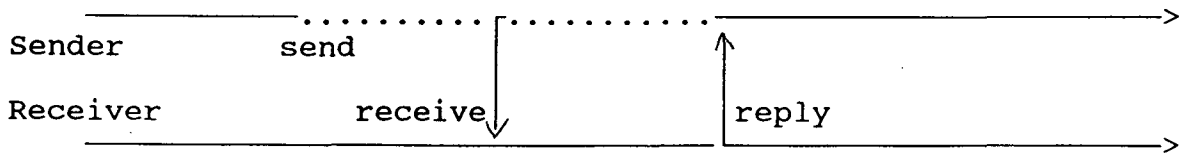
(a) Unconstrained Send



(b) Send blocked until reply



(c) Unconstrained send



(d) Send blocked until reply

The unconstrained **send** and **receive**, and the **send** with blocking until the reply is received are shown in figure (a) and (b). In both these cases the **receive** is being executed before the **send** i.e. the receiving process has to wait until the **send** primitive is called by the sending

process. In the figures (c) and (d) the send occurs before the receive operation is taking place. In the figure (d) the sending process is delayed due to two reasons; firstly because the receiving process is not ready to accept the message , and then because the reply has to be returned before the sender can continue. This type of message passing where send is blocked until a reply comes is also called synchronised message passing.

2.5 Concurrent Languages

There are a number of languages providing concurrent statements with different types of communication and synchronisation facilities. Some of the languages supports synchronous message passing, asynchronous message passing, rendezvous, remote procedure calls, multiple communication primitives, operation invocation on objects and atomic transactions. Languages of the first two classes provide point to point messages. Rendezvous based languages support two way communication between senders and receivers. A remote procedure call is a two way interaction but its semantics are closer to normal procedure call. Languages in the fifth class use variety of one way and two way communication primitives. Object based languages also support one or more of the above primitives. Here unlike other languages communication is between objects rather than processes. As object encapsulates both data and behaviour, these languages may also be thought of as providing some form of data

sharing. Occam, NIL, Ada, Concurrent C are some of the important languages which are widely used nowadays.

Occam is modeled on C.A.R. Hoare's CSP (Communicating Sequential Process) and was designed for programming Inmos's transputer. Occam is essentially the assembly language of the transputer. The language uses synchronised message passing for its communication purpose. *NIL* or Network Implementation Language is a high level language for the construction of large reliable distributed software. NIL was designed by Robert Storm and Shaula Yemini at the IBM T.J.Watson Research Center. NIL is a secure language based on asynchronous message passing where one program module cannot affect the correctness of other modules. Security in NIL is based on an invention called *typestate*. The language *Ada* was designed on behalf of the Department of Defence by a team of people led by Jean Ichbiah [U.S.Department of Defense 1983]. Parallelism is based on sequential processes called task in Ada. Each task has a certain type, called its task type. A task consists of a specification part which describes how other tasks can communicate with it , and a body , which contains its executable statements. Tasks can be created explicitly or can be declared but in neither case it is possible to pass any parameters to the new task. Limited control over the local scheduling of tasks is given by allowing a static priority to be assigned to task types. There is no notation for mapping

tasks onto processors. *Concurrent C* extends the C language [Kernighan and Ritchie 1978] by adding support for concurrent programming. The language is developed at AT & T Bell Laboratories by N.Gehani et al. This language uses synchronised message passing for communication between processes. More about this language will be discussed in subsequent chapter.

So, from the above discussions it can be seen that the main reasons for writing an application in concurrent program are; high speed through parallelism, high reliability through replicaton of processes and data, functional specialization. Concurrent programming is welcomed by the designers of programming languages based on paradigms like logic programming, functional programming and object oriented programming. They reliazed that parallelism might be the solution to the problem of obtaining an efficient implementation of their languages. This has led to the development of several languages providing higher level of abstraction. In this chapter the communication mechanism for concurrent program has been narrated breifly ,in the next chapter the problem of message passing discrepancy for synchronous communication model of communication is illustrated.

CHAPTER THREE

MESSAGE PASSING DISCREPANCY

In developing software system it is seen that disassembling a program to subparts is generally not a difficult task but the hard part is to put the tasks back into coherent whole. Specially it is more difficult when developing concurrent programs, where the different subparts are dependant on each other and need to communicate with each other. Two types of communications are possible between different processes of concurrent program, they are by using shared data and through message passing. However there is some limitations in using shared data communication. One limitating factor is the performance degradation due to memory contentions which occur when two or more processes attempt to access the same memory unit concurrently. Another limiting factor is processor memory interconnection network itself. So, message passing techniques for communication between processors is becoming popular. The main advantage is one need not have to use global memory. Reliability of the system also increases with the use of local memory. Many languages such as Occam for Transputers, Concurrent C support message passing method for inter process communication. Message passing between different processes can also be done in two methods. One is asynchronous or buffered communication another is synchronized or communication through blocked **send** and **receive**. Buffered communication is the technique where a process sending a data is allowed to leave it in the

communication module for subsequent collection by a receiving process. Unlike buffered communication in synchronized communication what happens is that the execution of two processes is aligned in the communication module to allow the transfer of data from one process to the other to take place. Synchronized communication between two concurrent processes require each process to issue matching **send** and **receive** operations which are then synchronized to enable one process to transfer data directly to another. In operational terms the first process that attempts the transfer is made to wait until the other is ready. Similar is the case for receiving process.

3.1 DISCREPANCY

So, it can be seen that there is one problem, that is of synchronisation between different processes when developing concurrent program. As described in the earlier chapter that when synchronised message passing primitives are used then, when a process wants to send a message, it must call a **send** primitive and has to wait till the receiving process invokes a corresponding **receive** primitive. This is also same for a receiving process. When the receiving process wants to accept any message it has to call the **receive** primitive and has to wait until there is a corresponding **send** operation taking place. So, there is a problem of matching in this type of communication. If the **send** or **receive** operations do not have a matching pair, the

corresponding processes will go into a continuous waiting state. This is explained by a simple code segment of three communicating processes below;

procedure Firstprocess

```
var C1,D1,m,l:integer;
    check,flow:integer;
    A1[100],B1[100]:Array of integer;
    p2,p3:processId;
```

Begin

```
C1=0;
read(m);
for i=1 to m
begin
  read(A1[i],B1[i]);
  C1=C1+A1[i]*B1[i];
end;
```

```
p2=Secondprocess;
p3=Thirdprocess;
```

```
send(p2,C1);
receive(p3,D1,l)
flow=C1*l-D1;
receive(p3,check);
if(flow<check)then write("Increment the valve outlet");
else if(flow=check)then
write("Keep valve position same");
else if(flow>check)then
write("Decrease the valve outlet);
```

End

procedure Secondprocess

```
var a,b,k,D2,C2:integer;
    p1,p3:processId;
```

Begin

```
p1=Firstprocess;
p3=Thirdprocess;

receive(p1,C2);
read(a,b,k);
if(k<100)then
D2=C2+a;
else D2=C2+b;
send(p3,D2);
```

End

```

procedure Thirdprocess
  var k,C3,D3:integer;
      A3[100],B3[100]:Array of integer;
      p1,p2:processId;

  Begin
    read(k);
    C3=0;
    for i=1 to k
    begin
      read(A3[i],B3[i]);
      C3=C3+A3[i]*B3[i];
    end
    send(p1,C3);
    receive(p2,D3);
    D3=D3+C3;
    send(p1,D3,k);

  End

```

Here we see the processes Firstprocess, Secondproces, Thirdprocess are interacting with one another and each of them is doing some specific task in a process control system. The result generated by one process is utilised in another process for its computation. The final output comes from the process Firstproces. But there is an error in the Firstprocess. This is not a syntactical error, but error in matching. As it is seen due to the wrong placement of send and receive statements all the three processes will go into waiting state. There is a simile between this waiting state and that found in deadlock situtation. In deadlock also a set of process waits for some event to occur by another process of that set with no one being able to administer that event.

Here the situation is somewhat different, the processes wait not due to nonavailability of resources but not being able to communicate with each other. The problem arises due to the discrepancies in message passing between different communicating processes. Message passing discrepancies can be again divided in two categories. The communication between processes can be of unpredictable order for some problems. Suppose there is a problem which is divided into different subparts where each part is being executed by a process. The requirement may be that each process will communicate the intermediate results generated by it whenever it gets any break. One situation may arise when all the processes get some solution at one time and tries to communicate with each other. Here all the processes will try to send the messages with no one willing to receive. Instantly all the processes will go on in infinite waiting state. The main reason for this problem is that there is no guarantee that a process will send exactly one message for each it receives. The number of incoming messages in a given period may be quite unpredictable as may the number sent, and the two numbers may be completely unrelated. This thing can be avoided by making the processes behave in a more regular and predictable manner. This disciplined approach is more reliable than the free for all approach, although it may not be so fast. But here also problem can arise. One case may be that all the processes begin to work by sending messages or status information, then there will be none willing to

receive any message and immediately all the processes will go on in continuous waiting state. For the system to run successfully at least one of the process must begin by receiving. This is an example of excess elegant symmetry. Many problems have regular structure which tempts the programmer to solve them using a number of similar concurrent processes. But if they are too similar then, when one attempts to send a message , so will all the others attempt to send, causing probably infinite waiting condition for some or all of the processes.

The problem just described above is of dynamic nature, which occurs during runtime, the programmer does not have any control over them. The problem requirements become such that the program becomes prone to this type of discrepancies. This shortfall can not be predicted during developing the programs. And as the length of program increases, the probability of this problem also increases. Another kind of thing can happen that is the discrepancy in message passing can creep in during the development of the concurrent program. The reason for which is erroneous coding during the development. The error which was described in the previous example of three communicating proceses P1, P2, P3 is of this type which has occurred due to improper synchronisation during developing the software.

At the time of developing the software the programmer will be aware of the different processes to

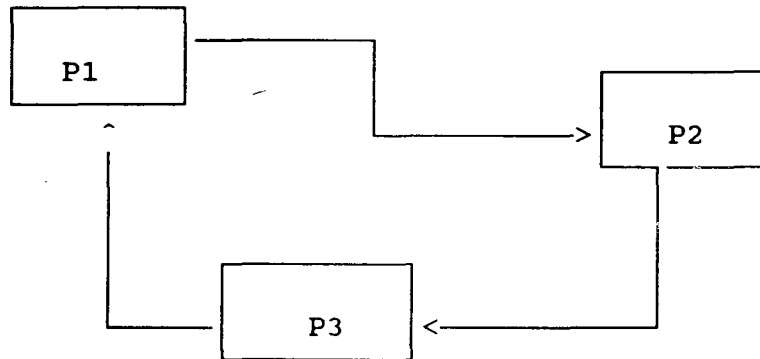
exchange messages. The intermediate results generated can be used by many processes for their computation and one process can use results of many other computing processes for its computation. Now the question comes that in which order the result will be send to those processes. Similarly for a receiving process the problem is to determine in which order it will receive different messages from different sending processes. Natural tendency of programming is whenever a result is ready, it will be send to the processes those which wants them for their computation and also similar is the case for receiving operations. Whenever a data will be first used only before that it will be attempted to receive from other processes. So, for successful message transfer it is necessary to keep the order of message passing proper. But it becomes very difficult with the increase in length of the program and number of interacting processes.

It can be seen that if the program is relatively small then this type of error can be detected by the participating programmers. They can make a table of interprocess message passing information for each process and can match them with the tables of other processes. Even for relatively small programs this will take quite a long time. For large programs with a number of processes and interprocess message passing , cheking all the processes for finding discrepancies will not only be a very time consuming job but also very error prone. For every message passing i.e

send or **receive** operation from a process the programmer has to check not only that the destination process has the matching **receive** or **send** but also that the corresponding process will be able to perform those operations i.e. he has to check that whether there is any discrepancies in message passing between that process and any other process, which process may eventually be dependant on the first process invoking the message transfer. So, it is seen that in a way the programmer has to check all the processes which are participating in message transfer between each other. As with large software, the entire program can be divided in different modules comprising of a number of processes and each module can be developed by different programmers, so the task for checking and detecting this type of errors manually will be very time consuming and there is every possibility of human error to creep in.

So, it can be realised that the problem of message passing discrepancies are more difficult to solve than that of deadlock due to resource sharing. In case of deadlock some prevention and avoidance mechanism can be used to cope with it. Even deadlock removal is not very difficult in the sense that what we have to do is first identify the processes in the deadlock cycle and then preempt some of the processes and provide the resources used by them for use by other waiting processes. The processes which will be preempted depends upon the policy which the system is following. The situation is not so straightforward in the case of message passing

discrepancies. To break the waiting state some of the processes are to be provided with the required messages. An example with the three processes P1,P2 and P3 is shown below.



Here P1 is waiting for a message from P2, P2 is again waiting for a message from P3 and P3 in turn on P1. To break the waiting state one of the processes, assume P2 is to be provided with the message . But P2 gets the message from P3 so P3 should provide the message. But P3 is again waiting for a message from P1 and unless it gets the message from P1 it can not go on for further computation and provide the message to P2. Because further computation of P3 depends upon the message received from P1 ,so it becomes impossible to break the waiting state during execution. So the solution of the problem at runtime is not possible in any simple way. The best thing can be done is to try to avoid the situation as far as possible. This can be achieved if proper checking of the program is done before execution.

3.2 Aim of the project

In the project work a model is provided using which mismatch errors for processes communicating by the help of synchronised message passing can be detected. This is an userfriendly software which will help the programmers in developing large concurrent programs. The software will relieve the programmer from the boredom of checking the processes for correct ordering of interprocess message passing. The time saved can be used by the programmer for other fruitfull works or checking other errors in the program.

The **significance** of this detection is that if it is not used the processes will go on in infinite waiting state during execution and there is no available runtime solution for this error. All the processes are to be rolled back and the root for the problem is to be detected. This will cause enormous loss of computing time. Another importance of the problem comes to light when concurrent programs are applied to real time systems. For some critical system such as space flight actual testing of the program is not possible and a problem such as this can cause total wastage of lot of time and money.

Another aspect of the problem is that in concurrent programming environment if a process goes on to continuous

waiting state this can cause erroneous termination of the program other than blocking the execution of the program. The following example will explain the case.

within TIME ? p.send(parameters):expr;

Here **p** is a process valued expression on the transaction **send**. **TIME** is a number indicating the duration of the transaction call for which the sending process will wait to receive an acknowledgement from the receiving process **p**. If the receiving process sends the acknowledgment within the duration **TIME** then the value returned by **p** will be the value of the timed call expression. Otherwise, the call will be withdrawn, **expr** expression will be evaluated and that will be the value of the timed call expression.

Now the process **p** may also communicate during its execution with other processes and suppose due to discrepancies in message passing it is waiting on some communication with other processes. What will happen? The process **p** will not be able to accept the call **send** and the **expr** will be evaluated and the result will be the value of the process valued expression. So the execution of the sending process will go on in wrong direction ultimately producing wrong results. The implication here is that if the process **p** directly doesnot affect the output then none will be aware of this execution error because the program will generate results, but which is wrong. If this

misleading result is not detected then the actual error in the program will not be corrected and the program will not be able to perform its intended job without any recognizable defect. The effect of this bug in crucial systems developed, spending lot of manhour, money and time will be serious.

The program developed in the project work is able to detect this discrepancies in message passing for fairly large number of programs written in **Concurrent C** language. It can be understood that the importance of parallel processing is increasing day by day. From few years from now concurrent programs running on parallel hardware of network of multiprocessors will be used profusely for all business and scientific applications. The objective of this project is to help the programmer in developing concurrent programs for genuinely parallel hardware.

In the next chapter a brief discussion of the **Concurrent C** language is provided which is useful for the implementation purpose.

CHAPTER FOUR

GLIMPSE OF CONCURRENT C

Concurrent C is an upward compatible extension of the C programming language, it provides concurrent programming facilities. It has been developed by N.H.Gehani and W.D Roome in AT&T Bell Laboratories. Concurrent C is based on the synchronous message passing model that has been described in the concurrent programming section of this report. The development of Concurrent C has been done keeping two objectives in mind;

a) To provide a concurrent programming language that can be used for writing programs on genuinely parallel hardware, such as network of multiprocessors or workstations.

b) To provide a test bed for experimenting with a variety of high-level concurrent programming facilities and distributed programming.

4.1 Interprocess Communication

In Concurrent C programmers define processes that communicate by synchronous message passing. Synchronous message passing primitives combine process synchronization with information transfer. Two processes interact first by synchronizing, then by transferring information, and finally by continuing their individual activities. This synchronization is called a rendezvous.

In simple rendezvous, the exchange of information is unidirectional, from the message sender to the receiver. However, many process interactions such as client process requesting service from a server process, require bidirectional information transfer, and hence require two simple rendezvous. The server performs the request, and then if necessary, does a second rendezvous with the client to give it the results of executing the request.

In Concurrent C extended rendezvous or transaction concept is used. An extended rendezvous allows bidirectional information transfer using only one rendezvous. After the rendezvous is established, information is copied from the process requesting the service, the client, to the server. The client process is then forced to wait while the server process performs the requested service. Upon completion of the service, the results, if any, are returned to the client, which is then free to resume execution. From the clients viewpoint, an extended rendezvous is just like a function call.

It can be mentioned here that like Concurrent C the programming language ADA is also based on rendezvous model, but there are important differences between the concurrent programming facilities in the two languages.

The reasons for choosing message passing model of communication in Concurrent C as stated by the developers

are;

a) Most interprocess interactions are synchronous: the client requests a service, and waits for it. This matches the synchronous model perfectly. Thus although the asynchronous model is more flexible, few people will use this extra flexibility.

b) A synchronous model can be implemented more efficiently than an asynchronous model. For example, an asynchronous model requires message buffers and a sizable message controller, data must always be copied into a message buffer and then out. For the synchronous model, data can be copied directly from the client process to the server process, without going through an intermediate buffer and the servers reply can be copied directly to the client. Thus the synchronous model saves space and time.

4.2 Process, Process type and transaction

A process definition consists of two parts: a type (or specification) and a body (or implementation). A process is an instantiation of a process definition. Each process has its own flow of control; it executes in parallel with other processes. The existence of a process definition does not automatically create a process. Instead, the programmer must create each process explicitly at run time. One can think of each process as having its own stack, machine registers, program counter etc. Most implementations will have some underlying scheduler that runs these processes on the

available processors. Concurrent C does not define the scheduling policy, except to say that the scheduling policy should be fair. The process type is the public part of the process definition. Only the information specified in the process type is visible to other processes. A process body contains the code (and associated declaration and definitions) that is executed by a process of that type; it is analogous to a function body, which it resembles. Details of the process body is not visible to other processes.

The extended rendezvous model has a **client** process, which initiates an interaction, and a **server** process, which waits for an interaction. The process type defines the kinds of extended rendezvous for which this process can act as server. Each kind of rendezvous is called a transaction. For each transaction, the process type defines the name of the transaction, the types of the arguments passed by the client, and the type of the value returned to the client. In extended rendezvous the client process is referred as calling a transaction to the server process, and the rendezvous itself as the transaction call.

Process types and transaction declarations:

A process type has the general format;

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

where parameter-declarations is a comma separated list of parameter declarations as in

```
process spec multiply(int num, intmax_size)
```

If a process has no transactions, the type can be written as

```
process spec process-type-name(parameter declarations)
```

Concurrent C processes synchronize and communicate by means of transactions. A process type must have a transaction declaration for each transaction for which this process can act as server. A transaction declaration is like a function declaration except that it is preceded by keyword **trans**, and that the parameter types are explicitly specified. The form is:

```
trans return-type tname(parameter-declarations);
```

This declares a transaction named `tname`, which returns a value type `return-type`; `parameter-declarations` is a comma-separated list of parameter declarations. The parameters represent the data that the client gives to the server; the return type is the type of the data that the server returns to the client.

The same transaction name can be used in several process types, and those transactions can have used in several process types. Thus, a transaction name is only meaningful in the context of a specific process type. In short, transaction names are to process types as structure member names are to structure types.

4.3 Process Bodies

A process body has the form

```
process body process-type-name(process-parameter-names)  
    compound statement
```

The process body specifies the C statements to be executed by each process of that type. Each process is sequential program components that runs independantly and in parallel with other processes. The compound statement in the process body can have automatic variables; each process of that type will get its own set of variables. Process parameters are used in the process body just as function parameters are used in function bodies. The types for the process parameters are given in the process type; they are not repeated in the process body. Values for the process parameters are supplied when each instance of this process is created. Process bodies can contain any legal C statement, plus several Concurrent C extensions, such as accept and select statements. Process bodies can call functions; the function is considered to be executing on behalf of that process. Any function can be called by a process of any type. The create operator is used to create a new process of the specified type with appropriate values for the process parameters. For example, given the declaration

```
process spec check(int max){..}
```

the expression

```
create check(1000)
```

creates a new process of type check, with 1000 for the parameter max, create returns a process value for this type, in this case a value of type process check. In a process body, the return statement terminates the process that executes it. This is equivalent to running off the end of the process body. A process can not return a value.

4.4 Process States and Process Termination

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

(i) A process becomes active upon creation and remains in the state while executing the statements specified in the corresponding process body.

(ii) A process becomes completed when it executes a return statement in its process body, or when it reaches the end of its body.

(iii) A process becomes terminated when it has completed and all the processes created by it have terminated or it executes a terminate alternative.

A process can also be terminated explicitly by `c_abort` function, i.e. the call `c_abort(p)` aborts `p`. Aborting an active or completed process forces it to become terminated.

4.5 Transaction Calls

A transaction call is the caller side of the transaction. The format is similar to a function call;

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

Process-value is a process valued expression designating a specific process. The type of the process must have a transaction named transaction-name, and the types of the arguments must match that transaction's parameter types. Like C function arguments, transaction arguments are passed by value. The transaction call expression has the type returned by the transaction. In general, a transaction call can be used whenever an expression of that type is allowed.

The calling process is delayed until the called process accepts the transaction. The called process is given the values for transaction parameters specified by the caller. The calling process then remains suspended until the called process returns a value, this becomes the value of the transaction-call expression.

accept statements

The **accept** statement is called process's side of a transaction. An **accept** statement has a form:

accept transaction-name(parameter-list)

compound statement

The **compound statement** is the body of the **accept** statement. An **accept** statement can only appear in the body of a process whose type has a corresponding transaction declaration.

Receive of Transaction call

For an **accept** statement, if a process has one or more transaction calls outstanding for a transaction named **t**, then **accept** statement for **t** accepts one of them immediately. Transaction calls are accepted in first-in first-out (FIFO) order. If there are no outstanding transaction calls for **t**, then the **accept** statement waits until such a call arrives.

Once a transaction call has been accepted, the **body** of the **accept** statement is executed. Within the **accept** statement body, the parameter names represent variables that are initialized to the parameter values given by the transaction caller. The scope of a parameter variable is limited to the body of the **accept** statement. To retain a parameter value beyond the scope of the **accept** statement

body, the parameter value must be stored in a variable with larger scope.

The calling process is delayed until the **accept** statement terminates by completing execution of its body or by executing a **treturn** statement of the form shown below;

treturn [expression]

The value of the **treturn** expression is returned to the calling process. The type of expression must conform to the result type of the corresponding transaction. If the result type is void, then no value is returned to the calling process, i.e. a **treturn** statement without an associated expression is used. After executing the **treturn** statement, the process containing the **accept** statement goes on to execute the next statement after the body of the **accept**, and the process issuing the transaction call becomes free to resume execution.

An **accept** statement can only be used in the process body, it can not appear in a function. This restriction is made so that the compiler will know the type of the processes executing the transaction calls and can verify that the processes have those type of transactions defined.

4.6 An example program

Here a simple example program is given to illustrate the Concurrent C language;

```
#include<stdio.h>
#include<ctype.h>

process spec consumer()

{
    trans void send(int c);
};
process spec producer(process consumer cons);
process body consumer()
{
    int ch;
    do{
        accept send(c) {ch=c;}
        if(ch!=EOF)
            islower(ch)?putchar(toupper(ch)):putchar(ch);
    }while(ch!=EOF);
}

process body producer(cons)
{
    int c;
    do{
        c=getchar();
        cons.send(c);
    }while(c!=EOF);
}
main()
{
    process consumer q;
    q=create consumer;
    create producer(p);
}
```

It is an example of conventional producer-consumer problem expressed in Concurrent C.

4.7 Delay statement

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

delay duration

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

4.8 Timed transaction call

The timed transaction call allows the **client** process to withdraw a transaction call if the **server** process named does not accept the call within the specified period. A timed transaction call is an expression of the form

within duration ?p.t(actual-parameters):expr

where **duration** is a floating point expression, **p** is a process-valued expression and **t** is a transaction name. If the process **p** accepts this transaction call within **duration** seconds, the value returned by **p** becomes the value of the timed transaction call expression. In this case the expression **expr** is not evaluated. otherwise the transaction call is withdrawn, **expr** is evaluated, and its value becomes that of the timed call expression. The transaction call is withdrawn automatically, Concurrent C guarantees that the server process never accepts a call that has been withdrawn by the client.

4.9 Nested Process

Concurrent C does not allow process to be syntactically nested within functions or within other processes.

The concurrency model in Concurrent C is based on the rendezvous model concept. Concurrent C can be used for a variety of applications;

1. To implement parallel algorithms.
2. To write genuinely distributed applications such as distributed databases.
3. To write real time programs.
4. To implement operating systems.

The next chapter explains the implementation detail of the project.

CHAPTER FIVE

IMPLEMENTATION

Concurrent C has been chosen as the language on which the detection schema has been applied. The reason for choosing Concurrent C is varied. Concurrent C is an extension of C language. The popularity of C language is the main reason for choosing Concurrent C. As Concurrent C is similar to C language so any one knowing C language will be in advantage in using it and as a number of people use C language today so there is every possibility that Concurrent C will become popular in future. Second and more obvious reason is that the model developed can detect errors for synchronized message passing techniques and Concurrent C uses that mechanism of message passing between different communicating processes. The software has been developed in C [Kernighan & Ritchie] language.

The detection schema is divided into two parts; the first one is QUEUE GENERATOR and the second one is DETECTOR.

5.1 QUEUE GENERATOR

This is the first part of the detection model. The sole aim of this portion is to generate queues corresponding to each processes. These queues will be then used in the second portion which is the DETECTOR.

In this section of the program the file containing the Concurrent C program will be opened for analysis. The

file is read till the end is reached. Each of the process in the file may be involved in a number of message passing transactions , either sending data to another process or receiving data from other processes. The **QUEUE GENERATOR** analyses each of the process and for every process it makes queues listing all the transaction names encountered in the process, and their characteristics i.e. whether it is a sending operation or a receiving operation and if it is a sending operation then the identity of the receiver. However the identity of the receiving process is not stored which will be explained later.

In the previous chapter various features of the Concurrent C language has been discussed. It can be seen there, that the Concurrent C program consists of three main sections. One is the process specification section, where various attributes of the process is described, among these are the parameters that the process will take, the transaction in which it will enter, the processes to which it will send data. As seen in the examples given there that in the process specification section no identity of the sending process is given when a process is receiving any transaction. So, in developing the queues in **QUEUE GENERATOR** for receiving process only the transaction details have been included. Similarly it can be seen that the sending process doesnot have the transaction name, the specification of it consists of only the identity of the receiving process. Analysing the process body of the Concurrent C program it can be seen that

each receiving process has a process identity which is different from the process name. This process identity is used in the sending process to send data to the receiving process. The receiving process uses **accept** statement for receiving a data from a sending process. An example of sending statement is again given here for convenience;

```
cons.check(k);
```

Here **cons** is the process identification name of the receiving process , **check(k)** is the message to be sent. In the receiving process this message **check(k)** will be received as;

```
accept check(k)
```

So, the **QUEUE GENERATOR** will look for this type of operations in the process bodies. It discerns the process identification such as **cons** as described above and places the message in the queue with the name of the receiving process. Similarly if it encounters an **accept** statement it places the transaction in the process queue and mark it as receiving operation.

The detailed explanation can be found from the pseudo code of the **QUEUE GENERATOR** which has been given below.

Now the data types which has been used and the implication of them will be discussed which is needed for better understanding of the pseudo code.

queue: It is a record with four fields. **transac** is the field which contains the name of the message encountered in the process bodies. **client** field stores the process identification of the receiving process to which the message containing in the process body is sent. **key** and **loop** are two integer fields that will be used in the **DETECTOR** portion of the program. The declaration of the queue is as follows;

```
struct queue{
char transac[20];
char client[10];
int key;
int loop;
};
```

pa: It is also a record type data type with two fields. One is the **procname**. It is a string which contains the name of the processes. **stadd** is a pointer which stores the starting address of each of the process queues that will be generated during analysing the process bodies.

```
struct pa{
char procname[20];
struct queue *stadd;
};
```

ident: This record type data type contains fields which are **bdname** and **mpname**. Both of them are strings. **bdname** is used for storing the names of the processes which are used in the process body. **mpname** is used for the process identification names which are used within the process bodies for message passing. It is declared as;

```
struct ident{
char bdname[20];
char mpname[20];
};
```

A number of subroutines are also used in the **QUEUE GENERATOR**, they are as follows;

giveword: This is used for reading the file which contains the the program written in Concurrent C. As only the reading of the Concurrent C program is necessary so the file containing the Concurrent C program is opened in the reading mode only. **giveword** takes the address of the string **name** and each of the time it is invoked it stores the next word encountered in the file in the string **name**. **giveword** also returns integer values according to the terminator encountered while reading the word as stated below;

- 1: if '.' is encountered.
- 2: if '(' is encountered.
- 3: if ')' is encountered.
- 4: if '{' is encountered.
- 5: if '}' is encountered.

A string of characters, which has been read till then, is stored whenever the above five characters is encountered or a new line or a blank, or a tab, or an EOF is encountered. Only for the five characters mentioned the specified values is returned and for all other cases any nonzero value other than the five above is returned. When EOF is reached a specific value is returned to indicate end of file. The filepointer attached with the file is used for all the reading operations. The prototype of this procedure is;

```
int giveword(char *);
```


initialise: Whenever a process body is encountered in the file of the Concurrent C program a process queue is generated. This is done by dynamic memory allocation. All the information regarding message passing for that process is stored in the process queue. Process queues are of the data type queue as stated before. **initialise** is the procedure which will initialise each of the process queues before its first use in the **QUEUE GENERATOR**. Length of each of the process queues is a predetermined value. This procedure initialises the **transac** and **client** field of all the elements to a specific string and the **sprocess** and the **key** field to zero. The prototype of this procedure is;

```
void initialise(struct queue proqueue[QMAX]);
```

Here **QMAX** is the length of each process queues.

check_process: This subroutine is used to determine whether the word returned by the **giveword** is a process identification name or not. This procedure uses the **prname** array for detecting a process identification name. In the **prname** array the **mpname** field stores all the process identification name for all the processes. According to the syntax of the Concurrent C program it can be seen that the process identification name terminates with a dot. So, if a dot is encountered as a terminator for the string returned by the **giveword** this **checkprocess** is invoked. The string returned by **giveword**, stored in the **name** string variable, is matched with

all the process identification names stored in the **mpname** field of the **prname**. If a match is found it returns a value accordingly. Prototype of this procedure is;

```
int check_process(char *);
```

s_addqueue: This is used for adding the messages encountered in the process bodies for the sending operations. It stores the operation names and the identification of the receiving process. The procedure takes the process identification name, the message name and the address of the next vacant element of the process queue of the sending process as input. It stores the process identification name in the **rprocess** field and the message in the **transac** field in the element of the process queue. If the message has been found within the loop then the **loop** field is marked to one. The procedure returns the address of the next vacant element of the process queue for the process for which it is invoked so that this address can be used for subsequent operations. Prototype of this procedure is;

```
struct queue *s_addqueue(char, char, struct queue *);
```

r_addqueue: This is the counterpart of the **s_addqueue** applied for receiving operation. Whenever an **accept** statement is seen in the Concurrent C program this subroutine is invoked and stores the message in the queue for that particular process. As stated in the **s_addqueue** this procedure also takes the address of the next vacant element of the process queue of the process where the **accept**

statement has been encountered. The message name is placed in the `transac` field of that element. If the message has been encountered inside the loop the `loop` field of the element is marked to one. The prototype of this procedure is;

```
struct queue *r_addqueue(char,struct queue *);
```

proarray: This is an array of record type `pa`. This array will contain the name of each of the process queues and the starting address for each of the process queues for each of the process. The starting address must be nonzero.

The algorithm for the **QUEUE GENERATOR** is described below;

ALGORITHM

1. Enter the name of the file containing the program written in the Concurrent C language.

2. Open the file whose name is entered for reading, assign a file pointer to the file for subsequent access purposes.

3. If the file cannot be opened, write **error in opening the file** and end the program.

4. Initialise the **procname** and **stadd** field of the **proarray**

Also initialise the **bdname** and **mpname** field of the **proname** array.

5. Get a word from the file invoking the **giveword**. If it is end of the file go to step 28.

6. Check whether the word is **process**; if then go to step 7; otherwise go to step 5.
7. Get another word from the file invoking **giveword**. Check whether it is **spec**; if not then go to the step 15.
8. Read a word from the file by **giveword**.
9. Check whether an opening paranthesis (is reached, if not go to the step 8.
10. Read a word from the file invoking **giveword**.
11. Check whether it is **process** or not. If it is not a process check whether a closing paranthesis) is reached, if then go to step 5; otherwise go to step 10.
12. Read a word from the file by **giveword** store it to the next element of the **praname** array in the **bdname** field.
13. Read another word from the file by **giveword** and place it in the **mpname** of the **praname** array.
14. Go to the step 10.
15. Check whether the word encountered is **body** or not. If not go to the step 5, else allocate a storage area for the process queue for that process of type **queue** and store the name of the process and the address of the process queue in the two fields **procname** and **stadd** respectively of the **proarray**.

16. Read a word through **giveword**. If the end of process body is not reached then go to 17 otherwise go to 5.

17. Check whether it is **accept** statement or not. If not go to the step 19.

18. Read another word and add that word through invoking **r_addqueue**. The procedure will return the address of the next element in the process queue for the process for which it is invoked. Go to the step 16.

19. If the word is process identification name go to step 20; otherwise go to the step 21.

20. Add the process identification name to the process queue. Read another word, this is the message name, store this also with the process identification name to the process queue using **s_addqueue**. The procedure will return the address of the next element of the process queue. Go to the step 16.

21. Check whether the word is any of the **for, do, or while**. Otherwise go to step 16.

22. Read a word through **giveword**. If the end of the loop has not reached, go to 23 otherwise if the end of loop has been reached then go to the step 16.

23. Check whether an **accept** statement is encountered or not if not go to the step 25.

24. Read a word through **giveword**. Use **r_addqueue** to place the word in the process queue for that process. Mark the loop field to one. Go to step 22.

25. Check whether the word is process identification name. Otherwise go to step 22.

26. Read another word from the file by **giveword**. Use **s_addqueue** to store the process identification name and the message to the process queue for that process. Mark the loop field of the element of the process queue to one. Go to step 22.

27. Close the file containing the Concurrent C program.

28. Display the total number of processes encountered and the name of them. Also display the process queues for all the processes.

29. Replace the process names in the **proarray** by the corresponding process identification names stored in the **mpname** field of the **proname** array.

The QUEUE GENERATOR portion of the model ends at the step 29. The output of this portion is the queues for each of the process found in the file of the Concurrent C program. These process queues generated in this portion will be used in the next portion i.e in the DETECTOR section for simulating the message passing operation. The pseudo code

for this algorithm is also given here for better understanding.

Pseudo-Code

```

BEGIN
write("Enter the filename containing the Concurrent C
program")

read(filename) /File name is a character string where the
               / name of the file where the Concurrent C
               / program is written is stored
fp=fopen("filename","r") /The file name is attached with the
                        /pointer type variable fp for
                        /subsequent reading purpose. The
                        /file opened for reading only

/This portion is for initialising the proarray and pronaame
with a maximum number of elements, PMAX/

for lt= 1 to PMAX
begin
  proarray[lt].procname="EMP" /Initialising procname field
                             /to EMP
  proarray[lt].stadd=0       /Initialising stadd field to
end                             / zero
for lt= 1 to PMAX
begin
  pronaame[lt].bdname="END" /Initialising bdname field
                           /to END
  pronaame[lt].mpname="END" /Initialising mpname field to
end                             /to END

  j= giveword(name) /giveword is called,it returns the word
                   /in the name variable
start:
  if(name="process")then /Checking whether the word is the
begin                       / start of process body or process
                           /specification section
  j=giveword(name)
  if(name="spec")then /Checking that an entry to the
                    / process specification section has
begin                       / been reached
  j=giveword(name) /j is an integer which stores the
                    value returned by giveword/
  if(j <> 2) /i.e. an '('is not encountered/
  repeat j=giveword(name) until (j=2)

  if(j <> 3) /i.e. an ')'is not encountered/
  repeat
    j=giveword(name)
    if(name="process")

```

```

begin
  j=giveword(name)
  pronaame[pid].bdname=name /Here in the bdname
                             field of the pronaame
                             array the word
                             returned by giveword
                             is stored/

  j=giveword(name)
  pronaame[pid].mpname=name
  increment pid /pid indicates the location in the
end / pronaame array
until(j=3)
end

else if(name="body")then /This indicates an entry to the
begin process body is reached/
  j=giveword (name)

  Q_add=Allocate(type queue) /Allocate some
                             / memory location
                             / for the process queue
                             / of type queue
  proarray[item].procname=name /The name and
                             / address of the
  proarray[item].stadd=Q_add / process queue is
                             /stored in current
                             /location of
                             / proarray
                             /Process queue
                             / initialised

  initialise(Q_add)

  j= giveword(name)
  while not(end of process body) do
  begin
  loop=0
  if (name="accept")
  begin
  j=giveword(name)
  Q_add= r_addqueue(name,Q_add,loop)
  end
  else if(j=1)
  begin
  i=check_process(name) /Checking whether the name
                             / returned is process
                             / identification name

  if(i<>0)
  begin /storing operations and message
  process_id =name / name for sending process
  j=giveword(name)
  Q_add= s_addqueue(process_id,name,Q_add,loop)
  end
  end
end

```



```

else if((name=for)OR(name=do)OR(name=while))
begin
loop=1 /entering within the loop body/
j=giveword(name)

while not(end of loop body) do
begin
if (name="accept")
begin
j=giveword(name)
Q_add= r_addqueue(name,Q_add,loop)
end
else if(j=1)
begin
i=check_process(name /Checking whether the name
/returned is process
/ identification name

if(i<>0)
begin /storing operations and message
process_id =name / name for sending process
j=giveword(name)
Q_add= s_addqueue(process_id,name,Q_add,loop)
end
end
j=giveword(name)
end
end / end of the body for loops
j=giveword(name)
end
end / end for process body
end
if not(end of file)
begin
j = giveword(name)
go to start
end

close(filename) /closing the file opened for reading
i=1
while (proarray[i].procname <> "EMP")
begin
j=1
done=false
while((praname[j].mpname <>"END") AND NOT(done))
begin
if(praname[j].bdname = proarray[i].procname)
begin
proarray[i].procname = praname[j].mpname
done =true
end
increment j
end
increment i
end
end
END

```

Pseudo-Code for procedures of Queue Generator

```
giveword ( name:array of character )
begin
let= getchar (fp)      /This is for reading character from the
                        / file using the pointer fp

while (let = whitespace characters AND NOT(end of file))
  let=getchar(fp)
  i=1
  name[i]=let /storing the first character in name
  increment i
  let =getchar(fp)

while((let<>whitespace characters)OR(let<>'('OR')'OR'{'
      OR'}'OR '.')OR let<> (end of file))
begin
  name[i]=let
  increment i
  let =getchar(fp)
end

if(let='.') return 1
if(let='(') return 2
if(let=')') return 3
if(let='{') return 4
if(let='}') return 5
if(let =end of file) return end of file character

end

initialise(proqueue[:array of queue)
begin
  for i=1 to QMAX
  begin
    proqueue[i].transac ="EMP" /Address Q_add is taken which
    proqueue[i].client  ="EMP" / is pointing to the first
    proqueue[i].key=0      / element of this array
    proqueue[i].loop=0
  end
end

check_process (p[:array of ident)
begin
  i=1 /it checks the word stored
  while (p[i].mpname <> "END") /in name for a match in
  begin /the mpname field of
    if (p[i].mpname = name) then /praname array
      return 1
    increment i
  end
  return 0
end
```

```

r_addqueue(p[:array of queue,Q_pointer:pointer to queue,
           lp:integer )
begin
  Q_pointer.transac=p /the message is inthe variable p
  Q_pointer.key=0
  Q_pointer.loop=lp /loop field is marked according to
                    /the value received
  increment Q_pointer
  return Q_pointer
end

```

```

s_addqueue(p1[],p2[]: array of queue,Q_pointer:pointer to
           queue,lp:integer)
begin
  Q_pointer.transac=p2 /the message is in the variable p2
  Q_pointer.client=p1 /storing name of receiving process
  Q_pointer.key=0
  Q_pointer.loop=lp /loop field is marked according to
                    /the value received
  increment Q_pointer
  return Q_pointer
end

```

5.2 DETECTOR

This is the second portion of the model developed. It performs the actual work of checking the processes for proper ordering in message passing. Here the actual message passing between different processes of the Concurrent C program is simulated and tested if there is any mismatch in the order of interprocess communication. It takes as input the queues generated by the **QUEUE GENERATOR** for each of the process in the program for simulation. For the simulation purpose a queue is being used which is named *run_queue*. The simulation is an iterative method testing each of the elements of all the process queues for every process. It starts from the first process queue whose name and address

has been stored in the proarray and returns to it at the last of every iteration. Each iteration goes on till there is any process in the proarray. In the iteration for every sending operation in the processqueue for every process it places the operation in the run_queue and waits for a matching receive. If the matching receive is found then the sending operation is removed from the run_queue. Otherwise the sending operation waits till a matching receive is encountered. For a receiving operation a matching sending operation in the run_queue is looked for, if it is found then the receiving operation is removed from the process queue of the process to which it belongs, otherwise the receiving operation waits till a matching sending operation is found in the run_queue. These operations for sending and receiving messages goes on till there is any message left in any process queue or the remaining processes are locked in waiting state for message passing. If there is no message left in any of the process queues of the processes then it is declared that all the processes have finished message passing operation successfully i.e. there is no mismatch in the order of sending or receiving messages for the interacting processes. Otherwise if there is some messages left in either the run_queue or in any of the process queues of any of the process and no matching sending or receiving operation is found for any of them then it is declared that some of the processes are waiting. The operation of the processes which are waiting is detected. And finally all the following item is displayed, the processes which are waiting, the operation

on which they are waiting and to which process they are to send the message or from which processes they are to receive the message.

Now the main items used in the DETECTOR portion is described briefly.

run_queue: This one consists of a record type data type node containing four fields, **transac, rprocess, sprocess** and **loop**. The first one stores the message to be sent, the second one the identity of the receiving process. In **sprocess** the identity of the sending process is kept and the **loop** field indicates whether the message is encountered within the loop or not. The declaration for this is given below;

```
struct node{
char transac[10];
char rprocess[10];
int sprocess;
int loop;
} run_queue[RQMAX];
```

Here RQMAX indicates the length of the runqueue which is used for simulation.

check_wait: This is a subroutine that checks whether any of the processes is waiting for message transfer or not. According to the value returned by this procedure the DETECTOR searches if all the remaining processes are waiting for message transfer. This procedure is called at the beginning of each of the iteration. It takes the address of the **proarray** as input and returns integer values.

```
int check_wait( char p[])
```

The algorithm for the **DETECTOR** portion is stated below;

ALGORITHM

1. Initialise the runqueue with a specific string in the **transac** and **rprocess** field and zero in the **sprocess** and **loop** field.
2. Take the first element of the process queue of the first process from the **proarray** whose elements of the process queue are not finished.
3. Check the key value of it .If it is one call **check_wait**.
4. If **check_wait** returns zero then go to step 5, otherwise go to step 15.
5. Take the address of the first element of the processs queue of the process selected. Test whether it is a sending operation or not; if not go to the step 7.
6. Find an unused element in the **run_queue**. Store the item in the **transac** and **client** field of the process queue in the **transac** and **rprocess** field of the **run_queue**. Mark the loop according to that in the process queue element. Store the item number of the **proarray** in the **sprocess** field of the **run_queue**. Mark the key field of the process queue element to one. Go to the step 14.

7. Check whether it is a receiving operation, if then go to the step 8; otherwise go to step 14.

8. Test the **run_queue** to find a match in the item of the **transac** fields of both **run_queue** and the element of the process queue. If a match is not found go to step 13.

9. Check whether entry in the **rprocess** field of the **run_queue** is similar with the process identification name of the current process selected and also whether the loop condition is same for both of them. If all the conditions are same then go to the step 10 otherwise go to the step 13.

10. Delete the sending operation from the **run_queue**. Mark the elements of the **run_queue** as unused. Delete the sending operation from the process queue of the process of that operation.

11. Check whether there is any operation left in the process queue of the process of sending operation; if not delete the process from the **proarray**.

12. Delete the receiving operation from the process queue of the receiving process. Check whether there is any operation left in the process queue, if not delete the process from the **proarray**. Go to step 14.

13. Mark the key field of the process queue element to one.

14. Take the next process from the **proarray** if any and go to step 5; if there is no process left go to step 28.

15. Select the first process from the **proarray**. Take the address of the first element of the process queue for this process.

16. Check whether it is a receiving operation, if not go to step 22.

17. Test the **run_queue** to find a same entry in the **transac** field of both the **run_queue** and the element of the process queue; if not found go to step 22.

18. Check whether entry in the **rprocess** field of the **run_queue** is similar with the process identification name of the current process selected and also whether the loop condition is same for both of them. If all the conditions are same then go to the step 19 otherwise go to the step 22.

19. Delete the sending operation from the **run_queue**. Mark the elements of the **run_queue** as unused. Delete the sending operation from the process queue of the process of that operation.

20. Check whether there is any operation left in the process queue of the process of sending operation; if not delete the process from the **proarray**.

21. Delete the receiving operation from the process queue of the receiving process. Check whether there is any operation left in the process queue, if not delete the process from the **proarray**.

22. Take the next process from the **proarray** if any and take the address of the first element of the process queue for that process and go to step 16; if there is no process left go to step 28.

23. Select the first process from the **proarray** if any ; otherwise go to step 28.

24. Take the address of the first element of the process queue of the selected process. Check whether the key field of the element is one.

25. Select the next process from the **proarray** if any; go to step 24, otherwise go to step 26.

26. If the key fields of the elements of the process queues for all the processes tested are one then declare the **processes are in waiting state**; otherwise go to step 28.

27. Display the processes which are waiting, from the **proarray**. From the process queue of each of the process display the operation on which they are waiting and to or from which processes they want to exchange messages. Go to the step 30.

28. Check the **proarray** for any process; if there is any process left go to the step 2.

29. Display **no mismatch error found**.

30. End.

So, it can be observed that the final output of the DETECTOR section is the display whether there is any discrepancies in message passing between different processes which are consisting the Concurrent C program as a whole. The proper ordering of the message passing statements is necessary for successful execution of each of the processes and also of the Concurrent C program. If any discrepancy between some processes are found then the affected processes are listed in the output. The statements over which the mismatch is occurring are also displayed.

Pseudo-Code

```

BEGIN
    / Initialising the run_queue /

for i= 1 to RQMAX
begin
    run_queue[i].transac="EMP"
    run_queue[i].rprocess="EMP"
    run_queue[i].sprocess=0
    run_queue[i].loop=0
end

start:
    item=1
    while (proarray[item].procname = "END" AND
           proarray[item].procname <> "EMP")
        increment item

    q= proarray[item].stadd /starting address of the first
                           non empty process queue is
                           taken

    if (q <> 0) then
        begin
            lock=0

            if ( q.key = 1)
                lock = check_wait(proarray)
            if (lock = 0)
                begin
                    while (proarray[item].procname <> "EMP")

```

```

begin
q = proarray[item].procname
if ( q <> 0)
begin
if ( q.client <> "EMP" ) /indicating sending process
begin
rptr =run_queue / rptr is a pointer which is taking
/the address of run_queue
while ( ( rptr.transac = "NIL" ) AND
(rpitr.transac <>"EMP") )
begin / searching for a vacant position in
increment rptr / the run_queue
end

rptr.transac = q. transac
rptr.rprocess = q. client
rptr.sprocess =item
rptr.loop =q. loop
q.key =1
end

if ( q.client ="EMP" ) / indicating receiving process
begin
rptr = run_queue /rptr is taking the address
/ of run_queue
while (( rptr.transac <> q.transac ) AND
(rpitr.transac <> "EMP") ) /searching
increment rptr /for a match on transac field

check =0
if ( rptr.transac = q.transac )
begin
if (rptr.rprocess = proarray [item]. procname) then
if (rptr.loop =q.loop ) then
check = 1
end
if (check =1) then
begin
rptr.transac="NIL" /as a match is found so the
/operations are being removed from
rptr.rprocess ="NIL" / the run_queue
rptr.loop = 0
itno = sprocess
increment proarray[itno].stadd / in the proarray the
/ starting of the process queue is
/incremented i.e deleting the previous
/ operation
k = proarray[itno].stadd
if ( k.transac = "EMP" )
proarray[itno].procname ="END"
rptr.sprocess =0
increment q
proarray[item].stadd =q
if ( q.transac ="EMP")
proarray [item].procname = "END"
end

```

```

    if (check = 0) then
      q.key =1
    end
  end / operations for the receiving case is ending here
end

  increment item
  end
end
if (lock =1)then
  begin
  while (proarray[item].procname <> "EMP")
    begin
      if ( q.client="EMP" ) / indicating receiving process
      begin
        rptr = run_queue
        while (( rptr.transac <> q.transac ) AND
              (rptr.transac <> "EMP" ) ) /searching
          increment rptr /for a match on transac field

          check =0
          if ( rptr.transac = q.transac )
            begin
              if (rptr.rprocess = proarray [item]. procname) then
                if (rptr.loop =q.loop ) then
                  check = 1
                end
              if (check =1) then
                begin
                  rptr.transac="NIL" /as a match is found so the
                                  /operations are being removed from
                  rptr.rprocess ="NIL" / the run_queue
                  rptr.loop = 0
                  itno = sprocess
                  increment proarray[itno].stadd / in the proarray the
                                                  / starting of the process queue is
                                                  /incremented i.e deleting the previous
                                                  / operation
                  k = proarray[itno].stadd
                  if ( k.transac = "EMP" )
                    proarray[itno].procname ="END"
                    rptr.sprocess =0
                    increment q
                    proarray[item].stadd =q
                    if ( q.transac ="EMP")
                      proarray [item].procname = "END"
                    end
                  end
                end
              increment item
            end
          end
        end

      item=1
      flag =0
      while ( proarray[item].procname <>"EMP")

```

```

begin
if (proarray[item].procname <> "END" )
begin
q = proarray[item].stadd / checking if all the
/ processes are in waiting
/ condition.

if (q.key =0) then
flag = 1

end
increment item
end

if (flag = 0) then
write( "processes are waiting on message passing ")
go to finish
end
end

```

```

item =1
check =0
while (proarray[item].procname <> "EMP" )
begin
if (proarray [item].procname <> "END" )
check = 1 / indicating there are processes
remaining for message passing
operations in the proarray
increment item
end

if (check = 1 ) then
go to start
else write("no discrepancy found")

```

finish: end of the code

END

```

check wait( p[] : Array of pa)
begin
item = 1 / indicating the first item of the proarray
cond = 1 / used for cheking whether all the processes
/ are in waiting state
while ( p[item].procname <> "EMP" )
begin
if ( p[item].procname <> "END" )
q = p[item].stadd
if (q.key = 0)
cond = 0
increment item
end
return cond
end

```

The software for the model has been developed in C language and implemented in Vax 11/780 machine. The program written in Concurrent C can be within one single file or the different process bodies can be separated in different files in which case the name of those files will have to be included in the main file. The software has been tested with quite a number of programs written in Concurrent C language. The results , advantages and some shortcomings of this model and implementation has been discussed in the next chapter.

CHAPTER SIX

CONCLUSION

In this project a schema has been provided for detection of message passing discrepancies that can be creaped in during the development of large concurrent software. The developed software has been tested for a number of Concurrent C programs. It has successfully checked programs containing processes from 10 to 20. The number of message passing operations within one process has been varied from 5 to 20. The program has also been tested for Concurrent C programs with process bodies in different files ,the method which is very popularly used in modern programming specially where different parts of the program is developed by different programmers.

The main advantage of this model is that it is not necessary to run the developed software in parallel processors. This is developed to run in uniprocessorss making it cheaper for use by any user. It will be very usefull where there is limited access to parallel processors. Parallel processors are costly machines and till today most of the organizations donot have parallel processors and also those posses it have it in small numbers. So, these machines are heavily demanded specially by scientific community. It is to be taken care of that one should get maximum benifit out of its utilisation. The users those who donot have those machines, have to reserve them for use against a considerable

money. So, if there is any bug in the developed program then that will cause a considerable loss of computing time as well as money. As described in the previous chapters the error in message passing can be enforced during developing the program. One has to be very careful in coding to avoid this type of error. But with large softwares and with a number of programmers participating in developing it, the assurance that there will be no discrepancy in message passing operations is not guaranteed. So, the model developed will help to debug this type of errors. As it can run in uniprocessors by simulating the message passing operation so it is cost effective also. It will also save the time in two ways. The software is not required to run in parallel processors so the programmers need not to depend upon parallel processors for checking their programs which will save quite a lot amount of time and also the checking is not required to be done manually that will also save time.

There are some issues of the developed software which are application dependant. The length of the process queues, the memory for which will be dynamically allocated each time a process body is entered, and the length of the runqueue for simulation will depend upon the Concurrent C program. The number of processes in the Concurrent C program and the interprocess message passing operation can vary randomly. The length of the runqueue has to be modified accordingly. Similar is the case for the length of the process queues. The amount of memory that has to be allocated

for each of these queues is dependant on the number of message passing operations found in each of the process bodies. The user has to modify the length according to his needs.

There is one shortcoming in the software developed in the way that it is not intelligent. It can be observed, when checking message passing operations within the loops. The software can check whether the corresponding message passing statements has matching loop conditions or not. But it cannot check whether the conditions for entering or coming out of the loop are proper or not. In fact this cannot be checked statically, some of these conditions are dynamic in nature i.e. can change from one execution to another. The programmer has to take the responsibility for the validity of the conditions put forth. Another thing could be that if the corresponding loop conditions doesnt match then also sometimes the program can be correct. For this reason in the actual implementation an alternative is used where a warning message will be displayed if the corresponding loop conditions doesnt match. Further improvement can be done in this area to make the software more intelligent. The testing for GOTO statements are also not applied, it is assumed that there is no need of using GOTO statements in a structured language such as Concurrent C.

So, to summarize the whole thing, in this project a schema has been proposed to check the concurrent programs written in Concurrent C language for proper ordering of interprocess message passing statements. The idea can be extended to check concurrent programs in other languages also which use synchronised message passing techniques. The application of concurrent programs and multiprocessors are increasing, so, it is hoped that the developed model will be very helpfull to the programmers who are writing concurrent programs.

BIBLIOGRAPHY

1. P. Brinch Hansen; *The Architecture of Concurrent Programs*, Prentice Hall (1977).
2. D. Whiddett; *Concurrent Programming for software engineers*, Ellis Horwood (1987).
3. L. H. Jamieson, D. Gannon, R. J. Douglass; *The Characteristics of Parallel Algorithms*, MIT Press (1987).
4. G. R. Andrews; *Concurrent Programming: Principles and Practice*, Benjamin/Cummings (1991).
5. D. Bustard, J. Elder, J. Welsh; *Concurrent Programs Structure*, Prentice Hall (1988).
6. W. M. Gentleman; *Message Passing between sequential processes: the reply primitive and the administrator concept*, Software Practice and Experience Vol 11, 1981.
7. J. Wexler and D. Prior; *Solving Problems with transputers : background and experience*, Microprocessors and Microsystems, Vol 13, No 2, March 1989.
8. A. Knowles and T. Kantchev; *Message Passing in a transputer system*, Microprocessors and Microsystems, Vol 13, No 2, March 1989,
9. N. H. Gehani and W. D. Roome; *Concurrent C*, Software Practice and Experience Vol 16(9) September 1986.
10. N. H. Gehani and W. D. Roome; *Rendezvous Facilities: Concurrent C and the Ada Language*, IEEE Transactions on Software Engineering Vol 14, No 11, November 1988.

11. K.A. Murray and A.J. Wellings; *Issues in the Design and Implementation of a Distributed Operating Systems for a Network of Transputers* , Microprocessing and Microprogramming, Proceedings of EUROMICRO '88, Zurich 1988 (24).
12. K. Hwang and F.A. Briggs; *Computer Architecture and Parallel Processing* , McGraw-Hill International (1987).
13. G.R. Andrews and F.P. Schneider; *Concepts and Notation for Concurrent Programming* , ACM Computing Surveys. 15(1), March 1983.
14. H.E. Bal and Tanenbaum et al; *Programming Languages for Distributed Computing Systems*, ACM Computing Surveys, Vol 12(3), September '89.
15. H.E. Bal; *Programming Distributed Systems*, Silicon Press, 1990.
16. A. Basu, S. Srinivas, K.G. Kumar, A. Parulral and L.M. Patnaik; *Message Passing Multiprocessor*, Proc. Fifth IEEE symposium on Parallel Processing, California, 1991.
17. A.K. Srivastava and S.C. Kshetramade; *PRESHAK: A Generic Tool to Implement Application Specific Message Passing Communication Kernels for Concurrent Machines* , Proc. Fifth IEEE symposium on Parallel Processing, California, 1991.
18. B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice Hall (1988).