A Tool
For
Protocol Validation

A C K N O W L E D G E M E N T

We express our deep sense of gratitude to all the persons who have helped and guided us in making project of ours a success.

First of all, we whole heartedly thank our teacher and guide **SRI. ASWATH KUMAR M.** who had the confidence in us to offer this tedious project. We also thank him for his invaluable guidance which helped us complete the project.

We sincerely thank lecturer **SRI.N.V.SUBBAREDDY** who evoked confidence in us and encouraged and helped us all along the way. We also thank **Dr. VENKATARAMAN P.** Asst prof., Department of E.C.E., I.I.Sc., for evaluation and criticism of the project.

We express our heartfelt thanks and gratitude to our beloved Principal **Dr. R.KRISHNAMURTHY,** who created a healthy working atmosphere. We also thank our department head **Prof. K. BASAVARAJU** who provided us sufficient computer time slots.

We are indeed indebted to Sri. **S.SHESHADRI,** incharge of computer center for the co-operation he extended throughout.

We thank one and all who have directly and indirectly helped us in completing this project successfully.

PROJECT ASSOCIATES

A B S T R A C T

The production of error free protocols or complex interaction is essential to reliable communication. This report presents a technique based on reachability analysis to detect the design errors in a design. The implementation of this **"PERTURBATION TECHNIQUE"** has been explained. The software detects design errors such as dead locks, unspecified receptions, state ambiguities and non-executable interactions.

A simple, easy to write language has been developed to give the specification of the finite state model of each process in the system. An user interface has been provided where in the user can see the contents of each node in the reachability tree in an interactive session.

Finally, we explain a parallel program where each node in the tree each perturbed in a parallel fashion. This has been simulated successfully on a **UNIPROCESSOR** system. Our design suites for loosely couple multi-processor system.

****

C H A P T E R 1

INTRODUCTION TO COMPUTER NETWORKS

This chapter gives a brief account of computer networks and the Network protocols.

1.1. INTRODUCTION

In this age of information theory, emphasis is given to faster, more reliable and economically cheap er ways of providing information. During the 2 decades since their invention, computer systems were highly centralized and usually within a large single room. The modern innovation and inventions paved way for a new approach to computing and information processing.

The merging of computers and communications has had a profound influence on the way computer systems are organized. The idea of a single computer doing all the job is becoming obsolete. These are relaced by large number of independent but interconnected, computers. These systems are called **“computer networks”.**

“Computer Network” is an interconnected collection of autonomous computers[4]. Two computers are said to be interconnected if they are able to exchange information, irrespective of the medium. The condition “**autonomous**” needs every computer in the network be functionally

independent i.e, one computer cannot dectate the function of the other.

Each processor in the network is called a **“node”** computer network. The nodes in newer computer network offer additional facilities. Like message switching or packet switching, routing, flow control, network monitoring and management.

An important aspect of communication network are, reduced delay, reduction of cost. And improvement in throughput. The main objectives of computer network are [1]

1) To provide sharing of resources such as information and process.
   Ex: Net work users located geographically apart may converse in an interactive session.
2) To provide interprocess communication.
3) To make the computer system fault tolerant and thus more reliable. If one processor in the network breaks down, another processor in the network can take its place.

4) To furnish centralized control for a geographically distributed system ex:- Intentory management, defence monitoring.
5) To provide distribution of processing functions Since processing is done where the raw information is generated, it serves transmission costs and thus errors.
6) To provide compatibility of dissimilar equipment and software.
7) To provide the users with maximum performance with minimum cost.

1.2. INTRODUCTION TO PROTOCOLS :

A network consists of a collection of interconnected nodes that permit exchange of information among each other. An orderly exchange of data requires that each node conform to some preestablished rules. A **"protocol"** establishes the set of rules and standards to exchange information between any two nodes. Any protocol consists of 3 elements [1]

1) Syntax or structure of data and control messages;
2) Semantics or set of control messages to be issued, action to be performed and responses to be returned; and
3) Timing, or specification of order of event executions.

Consider the exchange of information between two processes A and B.

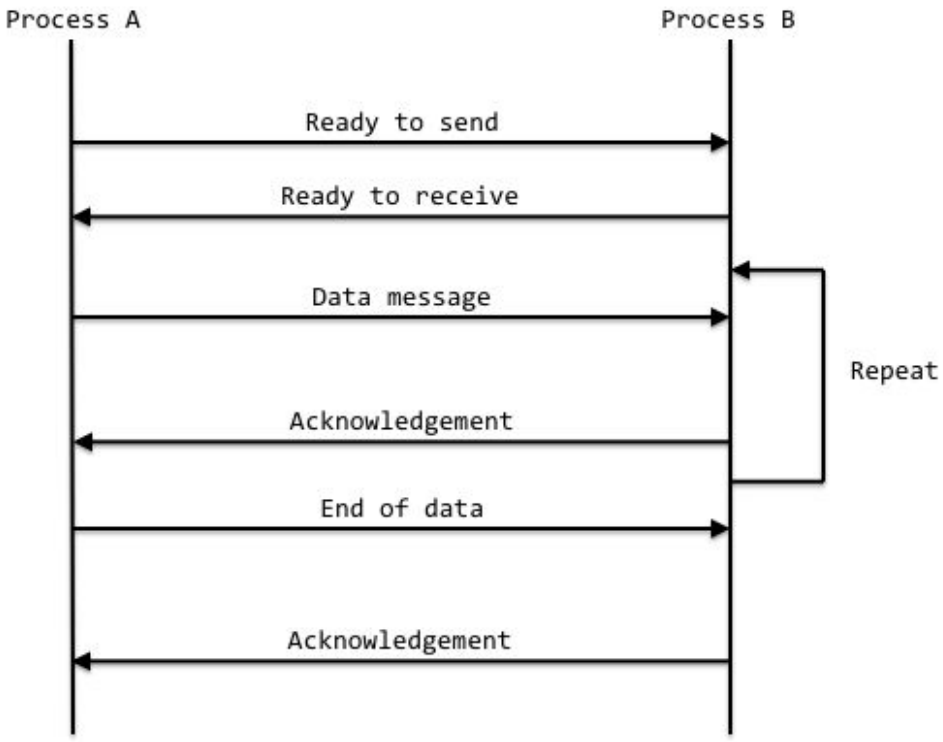


Figure 1-1

As shown in the above figure, process A first sends a ready to send control message. Process B responds by transmitting a ready to receive control message to process A. Then data messages are sent by process A and acknowledged by process B. In order to terminate the exchange, process A sends an end of data control message to process B and acknowledged by process B.

The control protocol for the above sequence of exchange messages must also specify format of each control message, header for data messages and order of messages.

1.3 FUNCTIONS OF PROTOCOLS :

As mentioned earlier, network nodes exchange two types of messages: control and data control messages are used to co-ordinate the exchange of data messages. The main functions of protocol are: [1]

1) Orderly exchange of data messages

2) Management of priority at both the network entry and transmission within network.
3) process synchronization

4) Session establishment between network users.
5) Session termination between network users
6) Reliable message transmission, including error control and recovery.
7) Optional packet switching through message segmenting and pipelining.
8) Resource management, monitoring and protection.
9) Sequencing of transmission of messages and delivery of messages.

1.4. LAYERING OF NETWORK PROTOCOLS :

There are three logical components that contribute a network protocol.

1) An entity such as application program residing in the computing system.

2) Some form of binding between three entities in different nodes, that provides a logical communication path between the two entities, and
3) A mechanism to transport information between the two entities that are paired.

Given the above 3 entities, the protocol designer must establish several rules for using and managing the transport network. These rules must be transparent to the network user and the communicating entities. Such an approach leads to layering of networking protocols. Any given layer logically exchanges message with corresponding layering of another node and processing at the other layers is transparent to it. A given layer also communicates with an adjacent layer above or below it through an interfacing protocol.

In an effort to standardize these protocols, a model was proposed by international standard Organization (ISO). This model is called ISO-OSI (Open System Interconnection) because it deals with interconnecting open systems.

OSI model consists of layers at each node of the network.

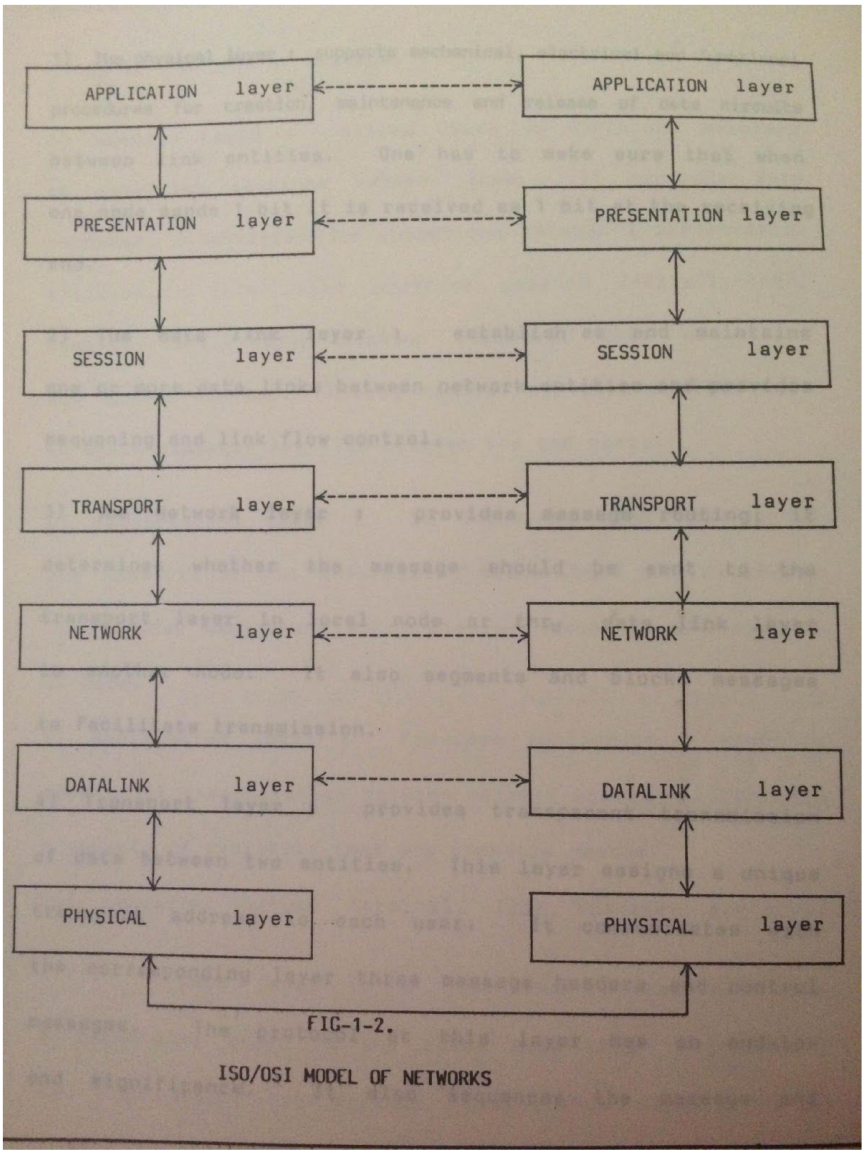


FIG-1-2.

ISO/OSI MODEL OF NETWORKS

1) **The physical layer :** supports mechanical, electrical and functional procedures for creation, maintenance and release of data circuits between entities. One has to make sure that when one node sends 1 bit, it is received as 1 bit at the receiving end.

2) **The data link layer :** establishes and maintains one or more data links between network entities and provides sequencing and link flow control.

3) **The network layer :** provides message routine; it determins whether the message should be sent to the transport layer in local node or thru data link layer in another node. It also segments and blocks messages to facilitate transmission.

4) **Transport layer :** provides transparent transmission of data between two entities. This layer assigns a unique transport address to each user. It communicates with the corresponding layer three message headers and control messages. The protocol at this layer has to an end-to-end significance. It also sequences the message and

Controls the flow of message.

5) **Session Layer:** allows users on different machines to establish sessions between then. It control data exchange, synchronises the operation between 2 presentation entities. This layer provides session initialization, session termination, session recovery, data delimiting and dialong control. Dialog control pertains to the order in which messages may flow betwenn the end users.

6) **Presentation layer :** is concerned with the syntax and semantix of the information transmitted and received. It includes management entry display, exchange and control of data.

7) **Application layer:** provides application specific aspects of communication between network users. It contains a variety of protocols that are commonly needed. Ex:- network virtual terminal, file transfer protocol.

Each layer in a node communicates with the corresponding layer in the other node through a protocol.

The processing is done in the lower layers are transparent to that particular layer.

1.5 MODELLING OF PROTOCOLS:

As mentioned earlier, in a layered structured communication networks, protocols are defined as a set of rules which govern the exchange of messages through instructions of partner of processes to achieve two goals.

a) to provide particular set of services to the layer above
b) to furnish a set of logical rules to its peer partner on the other side.

Our model of protocols employs a simple and commonly used representation of communicating processes. Each process is considered to be a finite state machine. Each pair of communicating

processes is connected by a full duplex, error free and FIRST-IN-FIRST-OUT (FIFO) channel.

Fig. 1-3 A simple communicating FSMs.

A simple communicating processes is illustrated in figure 1-3. A minus (-) sign indicates transmission of a message and a plus (+) sign indicates reception of a message. When a process is in a state from which there is an arc with negative sign, then it can traverse the arc and enter into the next transmitting message to the destination process via the connecting channel.

The destination process should be explicitly specific along which the message to be transmitted (Since only in a network consisting of 2 nodes the destination is implied, we re explicitly specifying destination processes which helps generalization).

If there is an arc which a positive sign, then the process can traverse that are only if the corresponding message has arrived on the channel. No assumption is made on the time as the process spends in a state before sending a message and on the time a message spends in a channel before it is delivered to the destination.

The figure shows the two processes use and server communicating with each other. Initially both are in their initial states **Ready and idle.** Users can send a request message REA to server. After receiving REQ server enters state SERVICE; when it is finished with the service, it goes back to the IDLE state by sending message DONE to the user. Between sending REA and receiving DONE user stays in the state WAIT.

While idle, SERVER finds fault with itself. If so it informs user about it by sending message ALRM. When USER receives on ALARM, it registers it and directs SERVER back to its IDLE state and the message ACK.

Formally we can define a finite state machine to be a 5-tuple (X,I,O,N,M)

Where

X : is a set of states;
I : is a finite set of inputs;
O : finite set of Outputs;
N : state of transition (N: IXX—X)
M : Output function (M:XXXI--O)

N and M express behavior of transmission.

As mentioned earlier, in our model each process is viewed as a finite state machine and the designing of the network involves the interaction between finite state machines.

We shall see in the next chapter that some design errors may creep in while designing a network and that the design needs to be validated. That is the purpose of this project.

****

2.1 INTRODUCTION :

The growing trend towards distributed systems and open communication network has greatly increased the complexity and sophistication of network protocols. To have proper communication, it is essential that protocols themselves be free fromdesign errors. Section 2.2 explains the design errors considered in this project.

A protocol can be verified either during their design phase before the system is implemented or during testing and simulation phase after the system has been implemented [5].

2.2 DESIGN ERRORS :

We have considered four types of design errors[5] that may occur in a protocol.

2.2.1. STATE DEAD LOCKS :

A state deadlock occurs when each and every process has no alternative but to remain indefinitely in the same state. In other words, a state deadlock is

Present when no transmission are possible from the current state of each process and when no messages are in channel.

2.2.2 UNSPECIFIED RECEPTIONS :

It occurs when a system can accept a message which is not specified in the design. Unspecified receptions are harmful since in the absence of adequate recovery routines. Occurance of anunspecified reception causes the respective process to enter an unknown state via a transition not specified in the design and the subsequent behavior of the system will be unpredictable.

2.2.3 NON EXECUTABLE INTERACTION :

A non executable interaction is present when a design includes message transmissions and receptions that cannot occur under normal conditions. A non executable interaction is equivalent to a dead code in a computer program.

Sometimes, interactions will be added to protocol for error recovery processes under abnormal conditions.
These are indicated by errors during validation. It is therefore suggested to validate a protocol under normal operating conditions.

2.2.4 STATE AMBIGUITIES:

Stable-state pair (X,Y) is said to co-exist when a state x in one process and a state y in other process can be reached with both channels empty. Then X and Y can co-exist until next transmission.

A state ambiguity occurs when a state in one process can co-exist stably with several

different states in other process. A state ambiguity need not necessarily represent a design error. But, protocol designer can confirm himself of the intention of design and the actual transition.

2.3. NEED FOR VALIDATION OF PROTOCOLS :

Validation of protocols have obvious advantages. We shall list most important ones [5]

1) Since protocols can be validated at an early

stage, unnecessary and incorrect implementation may be avoided. This reduces the cost of protocol development and testing.

2) Protocol designer do not usually for see all the syntactic properties of design. By validating the incompleteness, logical inconsistency and design errors in protocol, if any, can be indicated.

3) Formal methods of specifying protocol have helped in development of automated validation tools, which provide false proof, cheaper and faster ways of finding design errors, if any.

2.4. METHODS OF PROTOCOL VALIDATION :

In this section two protocol validation techniques are discussed.

2.4.1. DIOLOGUE MATRIX METHOD:

Diologue matrix theory [1] address the validation of a protocol between two asynchronous processes. A 2-tuple, containing a path from initial state to other states and back to initial state in each process, is called a dialogue [Ahuja].

The behavior of the protocol is incorporated in a validation function, by defining a number of fundamental rules. It is then applied to an interaction sequence to determine errors, if any.

The limitations [3] of this method are
a) It is restricted to protocol in which the interacting processes must return together to their initial states after finite number of steps.

b) It is limited to protocols defined between to processes.

2.4.2. THE PERTURBATION ANALYSIS:

To use this method it is essential to have a finite state model of a protocol. It explores all possible interactions of communicating processes by exhaustively generating all global states reachable given initial global state. Each global state consists of the states of all the processes in the system and contents of each channel in the system [5].

Each new global state is generated by reception or transmission of messages and global states are analyzed to check whether it represents any of the design errors. The sequence generated is represented as a reachability tree.

MERITS OF THE METHOD:

1) It allows the whole procedure to be automated [2]
2) The detection of design errors is straight forward and simple [2]
3) It can be implemented to a protocol for any number of processes [5].

LIMITATIONS :

1) When the protocol to be verified is fairly large, the time and memory required by this technique is very long and large[2].

2) The size of the reachability tree grows rapidly with increase in the number of processes [2].

4) Termination of the analysis is guaranteed only

All the channels are bounded.

2.5. CONCLUSION :

With all its limitations, it was felt that the perturbation analysis is the better technique, mainly because it offers a generalized solution to the problem of protocol validation. Hence, we

decided to implement the perturbation technique for the purpose of protocol validation and that is the topic of our discussion for the rest of our report.

****

C H A P T E R III

PROTOCOL VALIDATION : PERTURBATION TECHNIQUE

3.1 INTRODUCTION :

In this chapter we describe technique to detect the presence of design and potential design errors in the specified protocol. The perturbation method is reachability analysis which explores all

possible interactions of communicating processes by exhaustively generating all global states reachable from a given initial global state. A global state of a system is defined as a combination of the states of the communicating processes and the channels connecting them. An initial global state is defined as a combination of all initial states of processes, with all channels empty [2].

3.2. THE TECHNIQUE AND ANALYSIS :

The perturbation technique is defined as the execution of a single transition in one of the processes in the system. Each step is referred to as perturbation since it represents the smallest change that can take place in the system at any instant of time [2].

For a given initial global state, all possible
transitions are exercised, leading to a number of new global states. Each new global state is analyzed to determine whether or not it represents any design error. The process is repeated for each of the generated states until no more states are

created. The termination of this process is guaranteed only if the number of messages in any channel is less than a constant.

Now, let's take an example and build the reachability tree.

Consider the finite state model of a protocol for a 2 process system given in fig. 3.1.

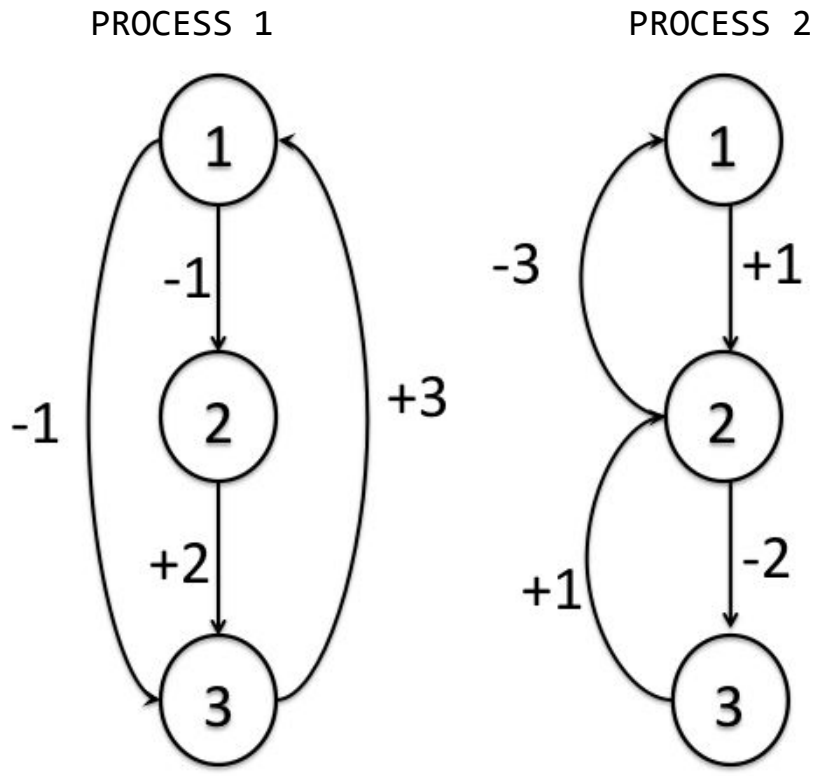


Figure 3-2 depicts the reachability tree for the example finite state model given in figure 3.1.

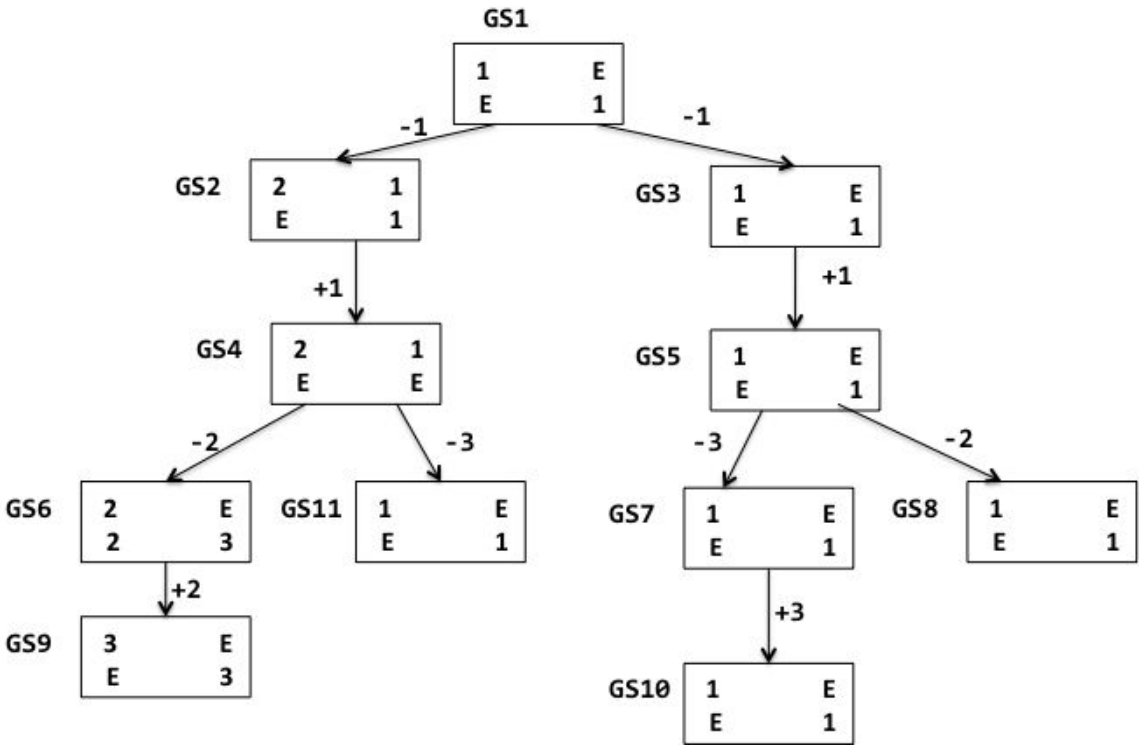


FIG. 3-2

Each global state is represented by a (2x2) array, where diagonal elements represent the current stable of each process and each off diagonal element represent the contents of the channel. Element is (i,j), i not equal to j, represent the messages in the channel P_i to P_j. An "E" represents an empty channel.

Building the reachability tree begins by defining the initial global state, GS1. The processes are in an initial state and all the channels are empty. GS1 is then **"perturbed"** into all possible successor states reachable executing (Transmission or Reception) a single transition either either in process 1 or process 2. Thus GS2 and GS3 are created by transmitting the message -1 on the channel process 1 to process2. The procedure continues by perturbing each of these new systems states in turn. **"New states"** means that only the global states which have not already been the trees.

This method has an attractive property that it creates the reachability tree for any n-process system interaction simply by defining the system state as an n x n array [5].

3.2.2. ERROR DETECTION VIA ANALYSIS:

In the example given we can see 3 types of design errors.

a)**Dead locks:** Consider the sequence of interactions

between the process1 and process2. Process1(p1) transmits the message -1 and process2(P2) acknowledges it. Then P2 transmits the message -2 and P1 acknowledges it Now, from this state, both the processes cannot transit to other states by transmission since there are is no arc transmitting message. Neither can accept since there are no messages on the channel. They remain in this state forever. This condition of the processes represent a dead lock.

b) **Unspecified reception :** Consider the sequence of interactions when P1 transmits message -1 and transits to state 3 from the initial state and the P1 acknowledges it. Then P2 transmits the message -2. Now both the process cannot transmit any message of P1 is not in a position to accept message 2. Even in this case, they wait for ever. This represents an unspecified reception.

c) **State ambiguity :** If all the channels are empty and subsequent system state exists, then a state ambiguity as occurred. In figure 3-2, GS1, GS4 and GS5 are all

representing state ambiguity. A state ambiguity need not be a designer error but it should be checked thoroughly whether it is part of design.

d) **Unreachable states:** are the states in finite state machines which are not reached by the corresponding processes. Examples of protocols having unreachable states are given in sample results.

3.3 CRITERIA FOR SELECTING THE LANGUAGE:

Following were the criteria for selection of the language.

1) the language should support handling of complex data structure.
2) It should have good text manipulating routines to create data base from input finite state machine specification.
3) Should support modular programming

Of all the languages, C language was seen to be the best. Since we're developing software on

UNIX, which provides a very good environment for software
Development in C, it was natural for us to go for C.

3.4. IMPLEMENTAITON OF PERTURBATION TECHNICQUE :

The following secions will explain different modules in the software.

3.4.1. SPECIFICATION OF PROTOCOL :

As described in section 1.5, our model of protocol is a **finite state machine (FSM).** The input FSM can be given in 2 ways. By directly specifying the transition by numbers in the menu provided by software. In the transition, we have to specify the message number, destination process and the next state to which the process goes to. Positive number indicates that the message is being received and negative number indicates the message is being transmitted.

The software by default assumes that the input is given from a file a language, which will be explain in subsequent paragraphs. To give the

input directly, we have to use –f option in command line while running software.

The other way is to specify protocol in a language. It must be remembered that the name of the input file must have a .prt extension, if it is not being given from standard input.

To simply the computational effort, it is assumed that the processes are assumed that the processes are numbered starting from zero. And so are the state numbers in the finite state machine.

The FSM in figure 3-1 thus could be modified as,

PROCESS 0 PROCESS 1

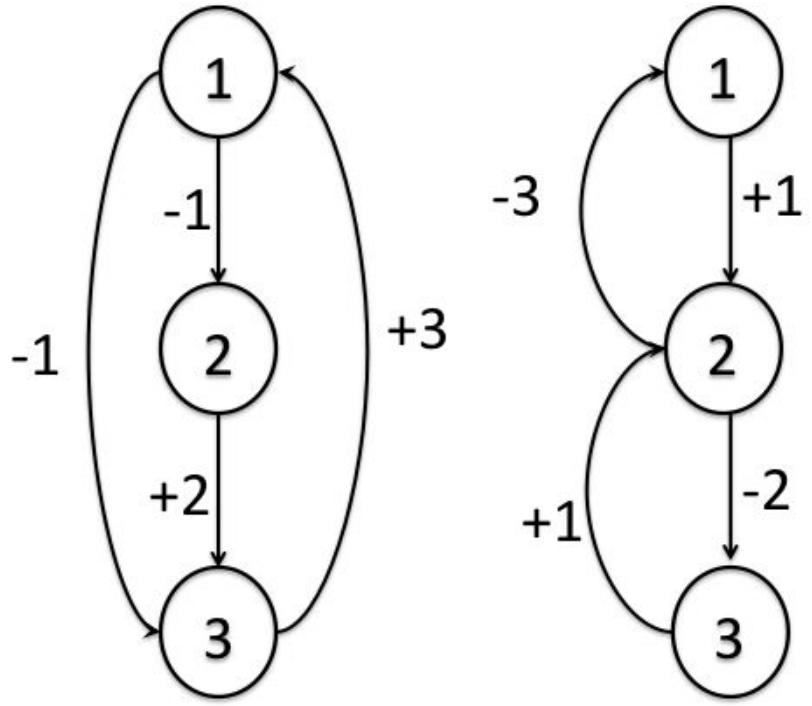


Figure: 3.3

The protocol specifications for FSM in Fig. 3-3 would be,

```
PROTOCOL example
Total processes    2
DEFINE  the_processes 0
Total stats 0 to 2
Initial state 0

BEGIN
     TRANS
          FROM 0
```

```
          TO   1
          When send (1,1);
     TRANS
          From 0
          To   2
          When send  (1,1);
     TRANS
          From 1
          To   2
          WHEN RECEIVE (2,1);
     TRANS
          From  2
          To    0
          When receive (3,1)
END
Define the process 1
Total state   0 to 2
Initial state 0
BEGIN
     TRANS
          From 0
          To   1
          When receive (1,0)
     TRANS
          From  1
```

```
            To    2
            When send (2,0)
        TRANS
            From  2
            To    1
            When receive  (1,0)
        TRANS
            From  1
            To     0
            When send  (3,0)
    END
```

Writing the above specification for FSM is self-explanatory. For example

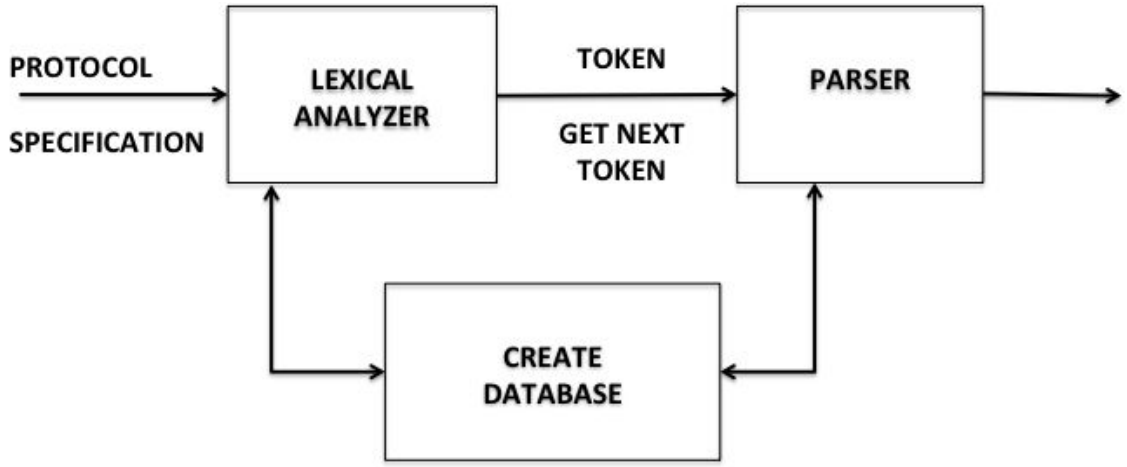
```
    TRANS
        From   1
        To     2
        When send (2,0)
```

Indicates that a process translate from state 1 to 2 by sending message 2 to process 0.

A simple translator is written which scans the input specification and creates a database as required by the software. Next section deals with the implementation of the translator.

3.4.2. TRANSLATOR :

Thte translator scans the input specification, checks for the specified syntax and duly reports the errors and error types, using the routine pr_error().



The lexical analyzer reads the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Lexical analyzer is used as a subroutine from the parser.

Since the software was developed on UNIX processing system, we had the privilege of using language development tool LEX*. LEX is a regular expression based lexical analyzer generator.

---

*Further details about LEX can be found in [LESK 75]

The recognition of expression is performed by a finite automaton generated by LEX. The program segments written by the user are executed in the order in which the corresponding regular expression

occur in stream. LEX generates the analyzer is C language.

In this translator, the scanner does not differente between upper and lower letters. Any occurance of non-numeric string, where a number is expected is considered to be a **“FATAL ERROR”** and the process is exited after informing the user above the occurrence of the error.

Syntax analyzer is implemented by the set of routines **create_data_base(), defined_process(), read_trans(), read_msg().**

create_data_base() control and co-ordinates the creation of the database. The database is created only if the input specification is free from errors.

defined_process() reads the definition of each process, read_trans() reads the definition of each transitions and the read_msg() routine to read the messagess.

Finally, if the specification is free from any errors, routine print_data() is called to create the database file. By default the data baseis created in an

unique file in /temp directory. The user of –d option makes the data base to be created in the working directory.

3.4.2.1. ADDITIONAL FEATURE OF TRANSLATOR :

Input protocol can contain comments between strings “/*” and “*/”.

The numbers may be replaced by strings sunch state (0), message(2), processes(0) by including the statement # include “state.h” at the beginning of the file. All the numbers may be replaced by strings but only by the prior definitons like the one below.

```
# define user 0
# define server 1
```

The input file is preprocessed using the ‘C’ preprocessor. Hance these strings will be replaced by numbers. The preprocessed output is stored under unique name in **“/tmp”** directory. Use of –P option will make the preprocessed output output to be stored in the working directory. This preprocessing facility increases the readability of the protocol specification.

If the user wants the file in his directory, he must use –d option in command line argument and give the filename in which data basewshould be stored.

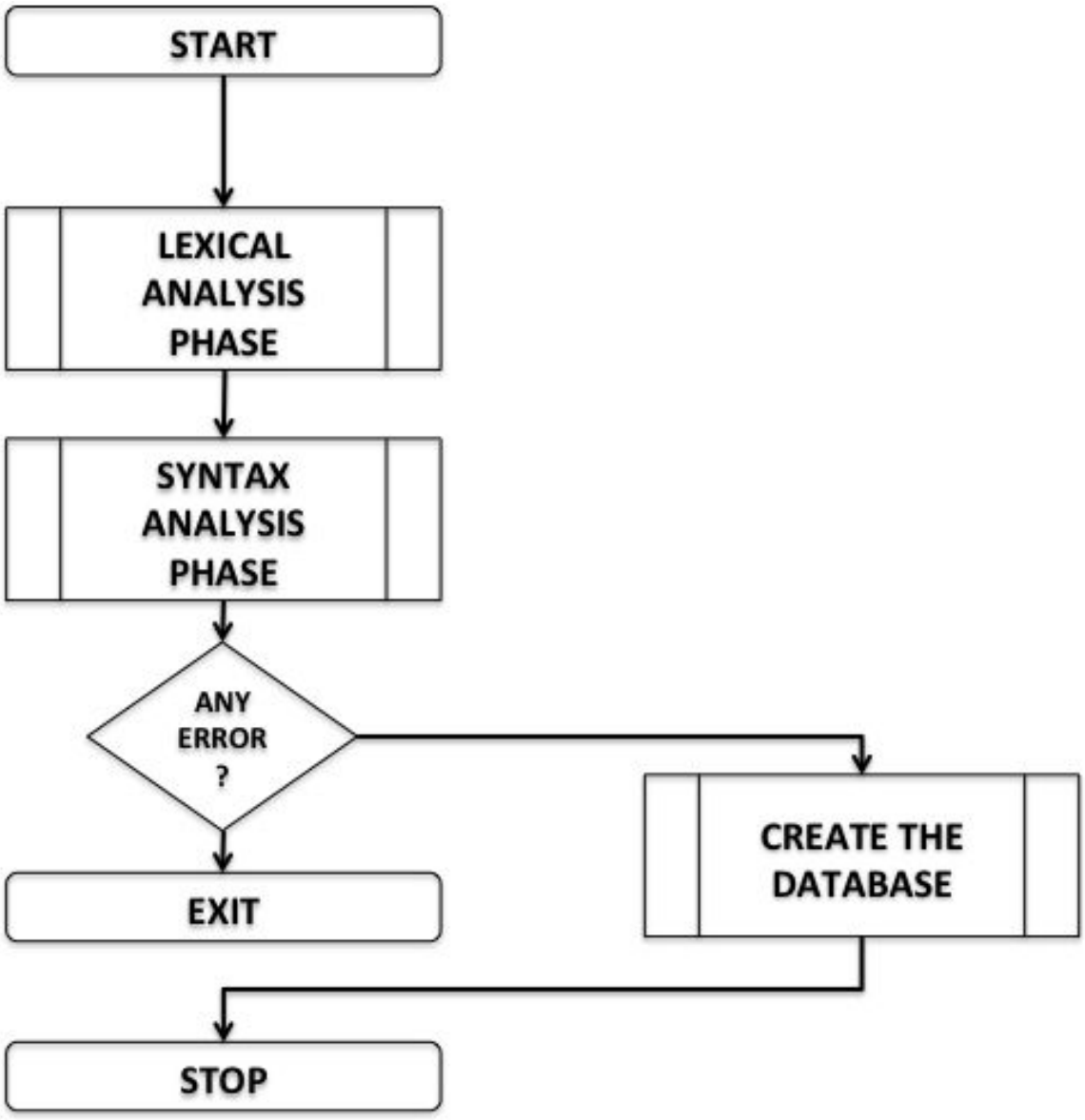


FIG. 3-4

FLOW Chart for creating database.

3.4.3 REPRESENTATION OF FSM:

The database is created in the following order.

1) Total number of processes.
2) For each of process,
    i) Process number under definition
    ii) Total number of states in the process.
3) For each state,
    i) Total number of transitions from that state.
    ii) Information about transitions in the order, message number, destination process and next state to which the process transits to.

The destination process number is always specified since only in a 2-process system, the destination is obvious.

3.4.3.1. DATA STRUCTURES :

The finite state machine is represented in the form of linked list. The following represent a node of structure fsm.

tot_state
process_number
init_state
st_pointer
fsm_pointer

A node in fsm

FIELD DESCRIPTIONS :

A)

(1) tot_state contains the total number of state in the FSM.

(2) Process number is an integer which contains the unique process number of the process under definition

(3) init_state contains the initial state of the process.

(4) **st_pointer** contains a pointer to structure state which contains the information about the transitions.

(5) **fsm_pointer** is a self pointer, pointing to the next node.

The structure state contains the following information.

tot_trn
reach_flag
inf
s_pointer

B)

(1) **tot_trn** contains the number of transitions rom the state to any other state

(2) **reach_flag** indicates whether that particular state has been reached or not. It helps in finding out unreachable states in the process.

(3) inf is a pointer to the structure trn which contains the information about transition of the state.

(4) s_pointer is a self-referencing pointer, pointing to the next node of the linked list.

The structure **trn** stores the following information.

msg
dest_proc
nst_state
n_ptr

C)

(1) **msg** contains the number of the message under transition

(2) dest_proc is the process under which the current process should interact with the message.

(3) **nxt_state** is the next state in the FSM that the process should change to after executing the transition.

3.4.3.2. ASSOCIATED ROUTINES :

accept() is the routine which accepts the data about FSM either from the database file created or directly if it has to accept data from the standard input i.e., keyboard, it displays a menu onto the terminal and accepts data from the keyboard. Otherwise data is accepted from the file pointed . Memory is allocated to the linked list as and when required.

get_fsm() is a routine which return a pointer to the structure state indicated by the argument proc_num and state_num. This routine in turn calls **get_state()** to return the pointer to the state.

set_reach_flag() routine sets the reach_flag in each state to 1. The state and the process are indicated.

3.4.3. BUILDING THE REACHABILITY TREE :

Before going into details of the routines for building the reachability tree, we shall consider the presentation of the tree structure in the computer memory.

3.4.3.1 DATA STRUCTURE:

Tree structure can be implemented in the natural way using self-referential data structures. Fig.3.5. shows structure of a node which represents a single global state in reachability tree.

node_num
msg
src_proc
is_next
process_status
ch
link_pointer

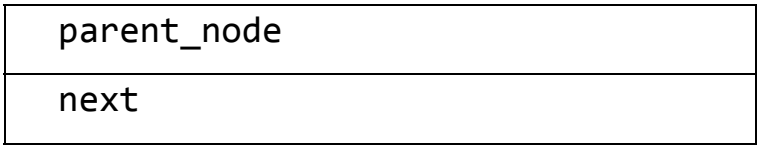


Fig. 3.5. A node of the reachability tree

FIELD DESCRIPTION:

1) **node_num** is an integer type field that stores the unique node number of the node. When there is an equivalent node in tree, node_num is set to NILL (-1)

2) **msg** is an integer type field that stores the message sent or received by the parent node in creating the node.

3) **src_proc** is an integer type that indicates the process which initiated the transition

4) **is_next** is also an integer type that indicates whether the node is a new node or it is already present in tree hierarchy or a new nodes is next is set to NULL (-1), else the node number of the equivalent node is stored.

(5) **process_status** is a pointer to the linked list process which contains states in which each process is, in the current node.

(6) **ch** is a pointer to the linked list channel which contains the list of messages in each channel.

Earlier it was state that global state can be represented in the form of n*n arrays. This has 2 disadvantages.

a) Inefficient usage of memory, since we have to define each channel to their maximum have to define each channel to their maximum capacity and some of the channels may remain empty.

b) It does not allow generalization of the problem since maximum dimension of the array has to be explicitly defined at compile time.

Hence, it was implemented in a linked list manner.

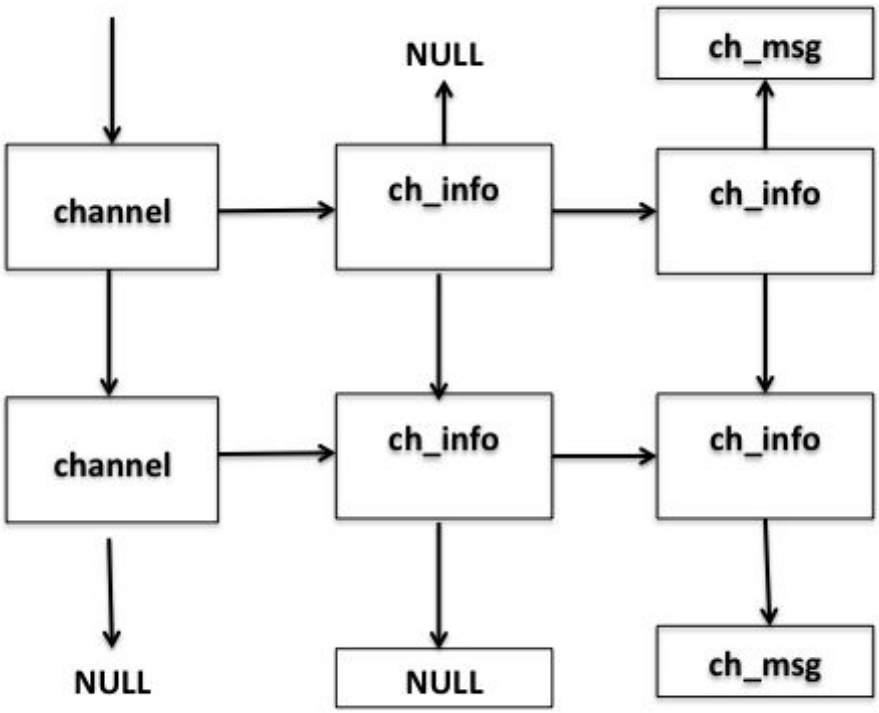


FIG. 3-6 CHANNEL for a 2 process system

Structure ch_msg contains the message numbers. Memory is not wasted since, we allocate dynamically. i.e., memory is allocated as and when message is put on the channel.

7) **link_pointer** will pointer to the node to which the node is equivalent if the node is a new node it will point to NULL.

8) **parent_node** is a pointer to the parent node of the node. It will left in tracking the pack which leads to design error.

9) **next** is a pointer to the linked list node_list which contains list of pointers to the child nodes produced. It can be seen that these pointers point to the some structure.

ROUTINES USED TO BUILT THE REACHABILITY TREE

initialize() is the routine which creates the **'root'** node of the tree structure. All the processes are initialized to states, specified by the input specification. All the channels are initially empty. Memory is allosed to all the channels, process lintel list.

build_tree() is the routine which builds the reachability tree. Initially the tree will have only the initial global state, i.e., the root. From this state, the child nodes are produced by trying all possible transitions. This is done by calling the routine **create_childnode()** for each node generated at the last level. The variable b_count stores the number of new nodes are generated. The tree built until no new nodes are generated. i.e., b_count = 0.

create_child_node() routine is the most important routine of the software. It takes a pointer to a node as an argument. Then for each of the process, it finds out whether it is possible to transit any other state

either by transmission or reception. To be able to receive a message for a process I from process j, the message should be the first in the queue in the channel j to i. Each time a transition is possible, a node is created and suitable changes are made in the state of the process and the channel affected.

At the same time reach_flag is set to the state created by the process which initiated the transition. After the node is created, it is checked whether an equivalent node is alredy present in the tree hierarchy. The routine **is_already_there()** does this job.

is_already_there() is the most beautiful routine of the software. It is designed to execute recursively so that comparison is made with all of the nodes of the tree starting from the root. Due to recursiveness of the routine, comparison is made starting from left to right. If the node generated newly does not match with any of the node in the tree hierarchy, it returns **NULL (-1)**, otherwise the node number which is matched is returned.

All the nodes generated are added on to the tree hierarchy but only the nodes which do not have an equivalent node is perturbed in the next level.

create_child_node() calls many routines to access data in lists and structures as shown below.

1) **get_process_status()** returns the state of a specified process in the list test given as an argument.

2) **get_channel()** returns a pointer to the linked list containing the message on a channel. The channel is specified as source process and destination process.

3) **get_new_channel()** just creates a new channel, copies the contents of the old channel and returns a pointer to the channel.

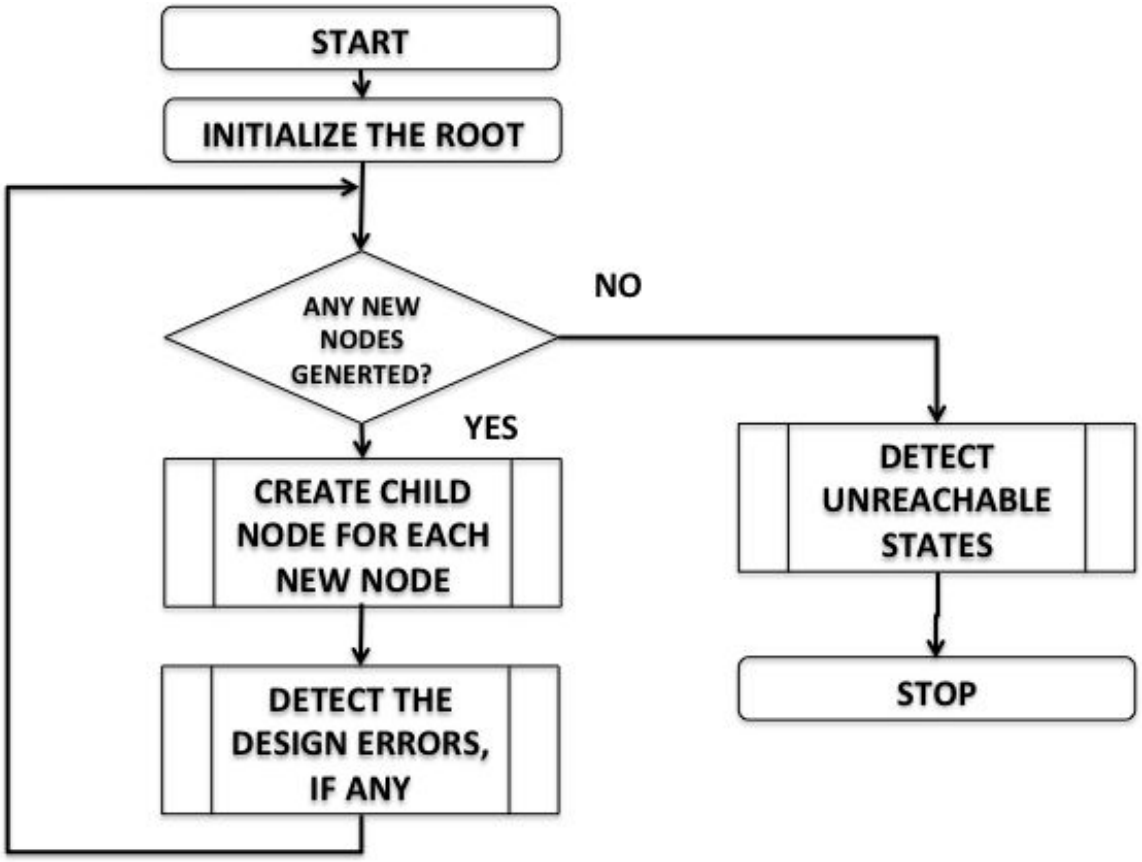
4) **print_channel()** prints the message in each channel. If any channel is empty, it is indicated by printing E.

To detect design errors, following routines are considered.

is_channel_empty() is a routine which returns a value 1 if all the channels are empty else it returns a value 0.

Each time when we try to perturb a new node, it is check for presence of design error. Error detection method is done as explained the section (2.2).

print_msg() is the routine which prints the status which are never reached in the reachability tree by the corresponding process. This routine is called after the full global tree is built.



FLOW chart for building the global reachability tree.

3.4.4. ADDITIONAL FEATURES OF SOFTWARE:

Along with providing clear input and output methods for the protocol, we have provided an user interface,

wherein the user can check the contents of each node in the tree structures. The facilities provided are:

1. User can at once see all the design errors in the protocol.
2. User can track any path in the tree.
3. User can see every useful information stored in each node of the tree structure.

To have an interactive session, the user must use –I option in the command line argument.

This user interface implemented in the form of windows. For this we have made an extensive use of **"curses"**[#] library available in UNIX. By typing options provided in the menu, user can have a neat dialogue with the system.

All the routines userd for the above purpose have been listed and are self-explanatory.

For details about the cursor library, refer UNIX system PROGRAMMING TOOLS LIBRARY.

3.5. EVALUATION OF THE SOFTWARE :

3.5.1. MERITS :

1) The perturbation technique has been successfully implemented.

2) The input FSM can be given in the form of text. It prevents the user from giving just numbers. Hence, it increases the readability of the input specification.

3) The user of C preprocessor further increases the readability of the input specification.

4) Since the errors are reported with good amount of detail, it is easy to correct syntax errors in the input specification

5) The software successfully detects the dead locks, unspecified receptions, state ambiguity and unreachable state.

6) The software provides a good user interface which allows the user to see the contents of the reachability tree, along with contents of each node is printed on a specified file.

7) Software is written to minimize the usage of memory at the cost of time. This was because, lesser the memory used, more nodes are generated, in a process involving more processes and messages. Even the runtime is not critical for protocol validation.

3.5.2. DEMERITS :

1) Time taken to build reachability tree is considerably long in case of more than 3 process system.

2) Some of the memory is wasted, only to provide, only to provide user interface which is not a part of the technique at all.

3.6. SUGGESTIONS FOR IMPROVEMENTS :

1) The design errors such as **“livelock”** cannot be detected by this software. Additional software can be written to detect these design errors.

2) Any method which improves time and memory will do a lot of good to the efficiency of the software.

3.7. OPTIONS AVAILABLE IN THE SOFTWARE :

Following are the options available in the software

-o file This specifies the output to be stored in the file name specified, rather than outputting ito to standard software.

-d file This specifies that the database should be created in the current directory rather than in **“/tmp”** directory.

-f specifies that the input being given directly as finite state machine rather than the default specification as a language.

-P file specifies that the C preprocessor output of the input FSM specification to be stored in the current directory under the name “**file**” rather than “**/tmp**” directory.

-I indicates that user wants to have an interactive session.

3.8. COMMAND TO GET PVT.

We have named our tool pvt as an abbreviation of protocol validation tool.

If the lexical analyzer is in the file pvt.l, type the following command to get lexical analyzer.

lex pvt.l

If the program is in the file pvt.c, type in the following command to get pvt.

cc –o pvt pvt.c –lcurses –ll

To run the software, type

pvt –o file.out file.prt

Where file.out is the output file and file file.prt is the input protocol specification.

****

C H A P T E R I V

PARALLEL PROGRAMMING FOR PERTURBATION TECHNIQUE

4.1. INTRODUCTION

4.2. DIFFERENT TYPES OF PARALLEL COMPUTERS

4.3. PARALLEL PROCESSING IN UNIX

4.4. IMPLEMENTATION

4.5. SCOPE FOR IMPROVEMENT

In this chapter we have explained the implementation of parallel programming for perturbation technique under unix which runs on a uniprocessor system.

4.1. INTRODUCTION :

For any work, there is always a better way. This was the inspiration for us, when we thought of modifying our software so that it runs more efficiently, taking lesser time.

Parallel programming is an efficient form of information processing which emphasizes the exploitation of concurrent events in computing processes [6]. Concurrency implies parallelism, simultaneity and pipelining. Parallel events may occur in multiple resources during the same interval; simultaneous events may occur at the same

instant; and pipelined events may occur in overlapped time spans.It is a cost effective means to improve the system performance through concurrent activities in the computer.

4.2. DIFFERENT TYPES OF PARALLEL COMPUTERS :

The most common and efficient parallel computer architectures are explained [6] in this section.

4.2.1. PIPELINED COMPUTERS :

Normally the process of executing an instruction in digital computer involves & major steps. **Instruction fetch** from main memory instruction decoding. Operand fetch and execution of the decoded arithmetic logic operation in a non-pipelined architecture, all these 4 steps should be over before the next instruction is fetch. In a pipelined architecture the successive instructions are executed in an overlapped fashion.

4.2.2. ARRAY COMPUTER ARCHITECTURE :

An array processor is a synchronous parallel computer with a multiple arithmetic and logic units, called processing elements [6]. These processing elements are designed to perform the same function at the same time in a synchronous manner.

Ex: Matrix multiplication, merge sorting.

4.2.3. MULTI PROCESSOR SYSTEM :

A basic multi processor organization contains more than one processor of approximately compatible capabilities. All processors have access to common sets of memory modules, I/O channels and peripheral devices. Most importantly, all the processors must be controlled by a single operating system providing interaction between the processors.

If the degree of the interaction is very high, then the processors are said to be tightly coupled. Otherwise, they are said to be loosely coupled.

4.3 PARALLEL PROCESSING IN UNIX

UNIX is a time sharing and multiprogramming operating system. It is time sharing since each user is allocated a quantum of time according to the priority, it is a time sharing system. Its capability on a uniprocessor system has rendered it the quality of multiprogramming.

Our aim is to exploit its quality of multiprogramming and execute different modules of the software simultaneously and exchange the data between the process. Since the amount of data exchange is small, our design suits a loosely coupled multiprocessor system.

4.3.1. DEFINITION OF A PROCESS :

A process is a program consisting of instructions and data and machine register contents, with an environment and an instruction pointer that indicates the next instruction to be executed. A process will always be in the processor queue until it is terminated.

4.3.2. SYSTEM CALLS USED FOR PARALLEL PROCESSING :

In our design, we mainly make use of following parallel processing features.

a. fork():

This is a system call, which causes the UNIX system to create a new process, called **child process**. Initially, the contents of the child process will be identical to that of the parent. The parent process data area, user stack will all be copied on to the child data area. The files will be stored in the child process. The **executing** program will be shared between the process. Once the child process is created, both parent and child

Process will be ready for execution and will be on the processor queue. i.e., 2 copies of the same programs will be running simultaneously.

Our aim is to run different modules of the program under different processes and save time. The fork system call returns the process ID number of the child process to the parent and a value of

zero to the child. Hence the execution of each process can be controlled by checking the value returned by the fork() system call.

b) pipes :

We have seen earlier that a single program can be executed uner different processes. Then there is a need for the interaction between different processes for exchange of data. Pipes facilitate this interaction by providing a path between different processes. For this purpose, we use the following system calls.

b1) mknod(): This creates a special **FIRST-IN-FIRST-OUT (FIFO)** pipe. Each pipe is named so that each process will read to write into corresponding pipe.

b2) open(): This system call open the pipe and returns an integer using which pipe accessing can be done. A pipe can be opened in **READ-ONLY, WRITE-ONLY** or both **READ-WRITE** node so that each process confines itself is specified operation.

b3) write() and read() : These system calls are used to read and write data from and to the pipe respectively. A write system call write data on to the pipe and returns. A read system call does not return if the pipe is empty, unless it is specified that it should return without waiting for another process to write onto the pipe, while creating the process.

4.4. IMPLEMENTATION :

We now, try to make use of the facility explained before for building the reachability tree and detecting design errors in a parallel fashion.

Building the tree is done by the routines

build_tree() and create_child_node() in the sequential program explained before. Therefore we need to make changes only in these routines.

Initially the tree will have only one node i.e., the initial global state for this we now create a single child process. Each child process

will have an exclusive pipe with the parent process for information exchange. The child process now tries to perturb this state into allpossible global states. If there is any possibility of any child being generated, the child node will write following data on to the pipe between that particular child and the parent.

1) Souce process which transmits or receives the message.

2) The message number.

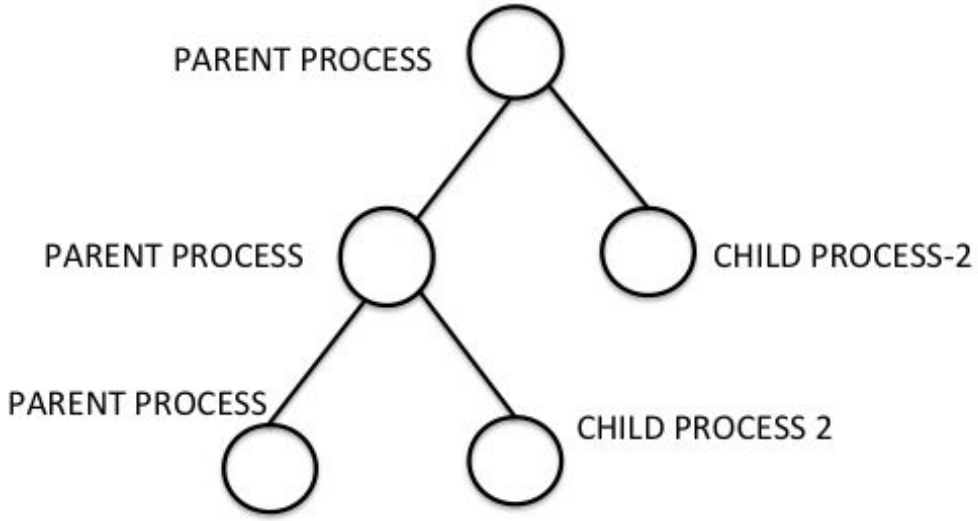
3) The next state to which the process transits to.

4) Information indicating the channel which should be altered after executing the transition.

End of the information about child nodes will be indicated by a source process numbered -1. This will be followed by a number which indicates whether any of the design errors are present or not in the parent node. A value zero indicates that there is no design error.

In the parent process the information about the transition and the child is read and the child nodes are created accordingly.

Consider an example where in we have to explore 2 global states. The parent process will create 2 child process where in each process will explore in global state.



State of the program after creation of 2 child processes.

The parent will now read the information from each child process. One by one and add on the child nodes to the structure.

Except the part of the child child nodes, the present software does not differ with the one presented earlier. But since, here each child node is created in a parallel fashion, it saves time.

UNIX system imposes the limitation that any process cannot open more than 20 files or processes. Hence, we've imposed a maximum limit defined by MAX_CHILDPROC. In case that a level has more than MAX_CHILDPROC nodes to be explored, we have taken care so that the loop will be repeated and all the new nodes are explored.

4.5. SCOPE FOR IMPROVEMENT:

The parallel processing environment provided under UNIX can be made best use of by implementing the following :

1) Instead of using pipes to transfer information, the tree structure may be build using a shared memory.
2) The amount of computation done in the child process can be increased so that the software runs faster.

****