Exploiting Symmetry for Analysis of Distributed Systems[1]

Vijay K. Garg[2]
Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712-1084
email: vijay@pine.ece.utexas.edu

Abstract

Distributed systems are difficult to design and the simplest of them can have subtle errors. Conventional automatic analysis techniques to catch these errors may be infeasible because the system may have a large, or even an unknown, number of processes. These techniques, which are based on state space exploration, run into the state explosion problem. Since most distributed systems have one or more sets of identical processes, we exploit the symmetry to reduce the state space for automatic analysis techniques. We describe a model called Decomposed Petri Net that facilitates such analysis. We present symbolic and induction techniques to analyze concurrent systems described using Decomposed Petri Net. We illustrate our techniques by analyzing several examples: 2-out-of-3 problem, dining philosophers problem and mutual exclusion problem. These techniques are applicable to systems that are configured either in a star topology or a ring topology. We also show how to extend these techniques for analysis of systems that use asynchronous communication. In particular, we show that for many problems, multiple unbounded reliable channels can be reduced to a single channel for verification purposes.

Index Terms Automatic verification, identical processes, symmetry, Petri Nets, concurrent systems, ring networks.

1 Introduction

Despite its economic benefits, distributed software has not gained wide acceptance mainly because of the difficulty involved in analyzing multiple processes. Research efforts in reasoning about programs can be divided into two groups - manual and automatic. Most researchers in distributed algorithms use manual reasoning based on the behavior of the system. Many proof systems have been developed for reasoning about safety and liveness properties [Apt 80, Hoare 85, Milner 80, Misra 81, Lamport 84, Chandy 88]. Manual analysis is error-prone and cumbersome; therefore we will restrict our attention to automatic analysis of distributed programs.

[1]A preliminary version of this paper appeared in 8th IEEE International Conference on Distributed Computing Systems, San Jose, CA 1988, pp 358-365.

[2]supported in part by NSF Grant CCR 9110605, Navy Grant N00039-88-C-0082, TRW faculty assistantship award, and IBM Agreement 153.

Automatic analysis of concurrent systems consists of computer exploration of all possible behaviors of the system. Many concurrent systems are based on finite state machines, making them particularly amenable to computer analysis. This approach has been used by many researchers, especially for the verification of communication protocols [Gerhart 80, Aggarwal 84, Blumer 86, West 78, Zafirolpulo 80]. There are many existing algorithms to check if a given temporal logic formula is true in the global state transition graph[Clarke 86a, Dill 86] There are two main hurdles to this approach - the number of processes may not be known initially, and the number of states may be too large. For illustration of difficulties involved in this approach consider the mutual exclusion algorithm in a ring network [Dijkstra 85, Clarke 86b]. If we know the number of processes initially (say 5), then we could construct the global state graph and check for any property in the graph. However, this approach becomes infeasible if the number of processes is not known initially or is large (say 100).

There have been many efforts to contain the state explosion problem. Many researchers [Dong 83, Kurshan 85] have studied this problem in the context of automatic protocol verification, where this problem is dealt with by collapsing multiple states into a single state while preserving properties that are important for verification. These efforts, however, are not very general and can be applied only for some specific properties such as deadlock-freedom. Moreover, they cannot be used for reasoning in networks with an unknown number of identical processes. Clarke et. al.[Clarke 86b] propose inductive techniques to prove properties of networks with identical finite state processes. Their approach consists of establishing a correspondence relationship between the global graph of n processes and the global graph of $n + 1$ processes. They show that if the correspondence can be established, then any formula expressed in Indexed Computation Tree Logic (ICTL) (a proper subset of CTL temporal logic) which holds in the initial state of a network with a small number of processes will hold for the network with a large number of processes. However, the step of establishing the correspondence is manual and could be difficult enough to defeat the original purpose of avoiding manual analysis. Our aim in this research is to minimize human involvement during the analysis. In their later paper[Clarke 87], they define a notion of closure of a process P denoted as P^*. Using this notion, they show that if two systems with r and $r + 1$ processes are equivalent under P^*, then any system containing more than r processes will also be equivalent for formulas expressed in ICTL. Again, the process of obtaining P^* is manual, and may require creativity from the verifier. Kurshan and McMillian [Kurshan 89] have also considered this problem. Their method requires finding a correct process invariant, and then checking that it satisfies the induction requirements (base case, and induction case) and the specification for which the system is being analyzed. If the invariant is known, then all above steps can be carried out manually. However, the process of finding an appropriate invariant is manual. Our techniques are much simpler, and do not require as much human involvement.

In this paper, we present algorithmic techniques for the analysis of distributed systems which have many identical processes. As most distributed systems have one or more sets of identical processes, these techniques have wide applicability, especially to the networks with a ***star***, or a ***ring*** topology. A ***star topology*** consists of a server (master) process and a set of identical client (slave) processes. Client processes interact only with the server process forming a star topology. This topology is common in centralized systems and various network servers such as name servers and printer servers. A ***ring topology*** consists of a set of identical processes communicating in a circular fashion. Each process has two neighbors and all messages originating at the process must go through either of the neighbors. This topology of processes is common for local area networks with token ring such as the Cambridge Ring Network [Needham 84].

All of the above networks show symmetry and it is desirable to have methods to reduce the global state space by exploiting this symmetry. We propose the following methods in this paper.

1. ***Symbolic Analysis*** : Instead of computing the actual global state, this method expresses the global state in terms of symbols. These symbols are then manipulated to compute other reachable global states. A symbol could stand for any unspecified component of the system such as the number of processes. With this method, one symbolic state represents multiple computed states thus reducing the state space substantially. We also show that when the number of processes is large, it may be more useful to treat the system as containing an unbounded number of processes. This analysis also cuts down the search-space significantly.

2. ***Induction Analysis*** : Instead of studying the system with a large or an unknown number of processes, this method analyzes it with a small number of processes and then the invariance of assertions is analyzed with the increase in the the number of processes. Since the analysis is done for a small number of processes, the reduction in the global state space is substantial.

This paper is organized as follows. Section 2 describes our formal model of processes called Decomposed Petri Net (DPN). This model facilitates symbolic and asymptotic analysis due to its ability of modeling multiple processes through multiple tokens, and its use of infinite number of tokens to model unbounded resources. Section 3 discusses symbolic techniques for analysis of systems expressed in the DPN with star topology. It illustrates these techniques for 2-out-of-3 problem. We extend these techniques for asynchronous communication. We show that multiple unbounded channels may be treated as a single channel for the purposes of logical analysis. Section 4 presents induction techniques applicable to a ring topology. Application of these techniques is shown to the dining philosophers problem, and the mutual exclusion problem in a ring. We also discuss how induction techniques can be applied when asynchronous messages are used. Section 5 makes concluding observations and discusses some future directions of research in this area.

We use calculational style of proofs for many of our theorems. A proof that $[A \equiv C]$ will be rendered in our format as

    A
$= \{$ hint why $[A \equiv B]$ $\}$
    B
$= \{$ hint why $[B \equiv C]$ $\}$
    C

We also allow implies ($\Rightarrow$) in the leftmost column. For a thorough treatment of this proof format we refer interested readers to [Dijkstra 90].

2 Decomposed Petri Net(DPN) Model

We first describe our formal model of computation. Since we are interested in automatic analysis, and most interesting properties for Turing-equivalent models are undecidable, we use Decomposed Petri Net model for our formal analysis[Garg 88b, 91b]. DPN is equivalent to Petri nets, but have the added advantage of possessing modularity. To decompose a Petri net, it is partitioned into multiple ***units*** which share the transitions of the Petri net. Each unit contains some of the places of the original Petri net. Intuitively, the decomposition is such that the tokens within a unit need to synchronize only with tokens in other units. A unit is a generalization of a finite state machine. Formally, a DPN (Decomposed Petri Net) D is a tuple (Σ, U) where

- Σ = a finite set of symbols called ***transition alphabet***
- U = a set of units $(U_1, U_2..U_n)$; each unit is a five tuple, i.e., $U_i = (P_i, C_i^0, \Sigma_i, \delta_i, F_i)$ where:
  - P_i is a finite set of ***places***
  - C_i^0 an initial ***configuration***, also called marking, is a function from the set of places to nonnegative integers $\mathcal{N}$ and a special symbol '*'. i.e.,$C_i^0 : P_i \rightarrow (\mathcal{N} \bigcup \{*\})$. The symbol '*' represents an unbounded number of tokens. A place which has * tokens is called a ****-place***. This component of a unit is the only one that changes as DPN executes. We use C_i to represent the configuration at any point of computation.
  - Σ_i is a finite set of ***transition*** labels such that $\Sigma_i \subseteq \Sigma$.
  - δ_i is a relation between $P_i \times (\Sigma_i \cup \{\epsilon\})$ and P_i, i.e., $\delta_i \subseteq (P_i \times (\Sigma_i \cup \{\epsilon\})) \times P_i$. ϵ represents null transition. δ_i represents all transition arcs in the unit.
  - F_i is a set of final places, $F_i \subseteq P_i$.

The configuration of a DPN can change when a transition is fired. A transition with the label a is said to be ***enabled*** if for all units $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ such that $a \in \Sigma_i$ there exists a transition (p_k, a, p_l) with $C_i(p_k) \geq 1$. Informally, a transition a is enabled if all the units that have a transition labeled a, have at least one place with non-zero tokens and an outgoing edge labeled a. For example, in Figure 1 ***get-item*** is enabled only if both p_4 and p_5 have tokens. A transition may ***fire*** if it is enabled. The firing will result in a new marking C_i' for all participating units. Based on this, we define the next-configuration function $\gamma_i : \mathcal{N}^n \times \Sigma \rightarrow \mathcal{N}^n$ for a Unit U_i (with $|P_i| = n$) as $\gamma_i(C_i, a) = C_i'$ where:
$C_i'(p_k) = C_i(p_k) - 1$
$C_i'(p_l) = C_i(p_l) + 1$.
$C_i'(p) = C_i(p)$ for other p.

Figure 1: A DPN machine for the Producer Consumer Problem

A *-place remains the same after addition or deletion of tokens.

As an example of a DPN machine, consider the producer consumer problem. The producer produces items which are kept in a buffer. The consumer takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. The consumer can execute ***get-item*** only if there is a token in the place p_4. Note how the *-place is used to represent an unbounded number of tokens.

We can extend the γ_i function to a sequence of transitions as follows:
$\gamma_i(\mu, t_j\sigma) = \gamma_i(\gamma_i(\mu, t_j), \sigma)$,
$\gamma_i(\mu, \epsilon) = \mu$
where ϵ represents the null sequence, and $t_j\sigma$ represents the sequence t_j followed by σ.

The ***language*** L of a Decomposed Petri net $D = (\Sigma, U)$ is defined as

$$L = \{x \in \Sigma^* | \forall i : C_i' = \gamma_i(C_0, x/\Sigma_i) : C_i'(p) = 0 \ for\ all\ p \in P_i - F_i\}$$

Thus, all the tokens in a final configuration must be in final places. The class of languages defined by DPN can alternatively be characterized by concurrent regular expressions[Garg 88b, 89, 91b]. These expressions are more powerful than regular expressions, and are defined using operations of choice (+), concatenation (.) , Kleene closure (*), interleaving ($\|$), interleaving-closure (α), and synchronous composition ($\Box$). As we will be using these operators in this paper for some of our examples, we define them below:
Let L_1 and L_2 be two languages defined over Σ_1 and Σ_2. Then,

- $L_1 + L_2 = L_1 \cup L_2$.
- $L_1.L_2 = \{x_1x_2 | x_1 \in L_1, x_2 \in L_2\}$
- $L^* = \bigcup_{i=0,1,..} L^i$ where $L^i = L.L...i\ times$
- Interleaving between two strings is defined as follows:
  $a\|\epsilon = \epsilon\|a = \{a\} \qquad \forall a \in \Sigma$

$a.s \| b.t = a.(s \| b.t) \cup b.(a.s \| t) \quad \forall a, b \in \Sigma, s, t \in \Sigma^*$
Thus, $ab \| ac = \{abac, aabc, aacb, acab\}$.
This definition can be extended to interleaving between two languages in a natural way, i.e. $L_1 \| L_2 = \{w | \exists s \in L_1 \wedge t \in L_2, w \in s \| t\}$ Note that similar to $L_1 \| L_2$, we also get $L_1 \| L_1$ denoted by $L_1^{(2)}$.

- Taking repeated interleaving of any language gives us its interleaving closure. Intuitively, the interleaving closure lets us model the behavior of an unbounded number of identical independent sequential agents. It is formally defined as $L^\alpha = \bigcup_{i=0,1,..} L^{(i)}$.

- $L_1 \Box L_2 = \{w | w/\Sigma_1 \in L_1, w/\Sigma_2 \in L_2\}$ where w/S denotes the restriction of the string w to the symbols in S. Intuitively, synchronous composition ensures that all events that belong to two sets occur simultaneously.

The definition of concurrent regular expressions and their relationship with Decomposed Petri nets is summarized in Appendix A. For further details of DPNs and its comparison with Petri nets [Peterson 81, Reisig 85], readers are referred to [Garg 88a,b, Garg 91].

In this paper, we will assume that all places in a DPN are final, and therefore the language L accepted by any DPN is prefix-closed, i.e. if $t \in L$, then all prefixes of t are also in L. We say that $s \leq t$ if s is a prefix of t, and $s < t$ if s is a prefix of t and not equal to t. We treat two processes identical if their language is the same. This requires an assumption of determinism in processes. For non-deterministic systems, the language accepted (or the set of traces) may not be sufficient and more general structures such as set of failures[Hoare 85] are required. Similarly, we have assumed the interleaving model of concurrency as opposed to the model of partial orders[Pratt 86].

We now describe methods to analyze processes described using DPN for various networks.

3 Symbolic Analysis: Star Topology

In these networks there is one server process that communicates with multiple clients. We assume that each of the clients has identical behavior. One of the advantages of the notion of ***token*** in a DPN is that it can represent a process; therefore, multiple identical processes are represented by multiple tokens in some state. If the number of processes is large or is unknown initially, we may use a symbol (say n) in a state to represent the unknown number of processes. Now we do the rest of the analysis in terms of these symbols. Alternatively, we may use $*$ instead of n, which would tell us the behavior of the system when n is very large.

We use symbolic analysis for networks with star-topology. A DPN representation of such a network would generally have two units - one for the server process and one for multiple clients. The multiplicity of clients is represented by the presence of multiple tokens in some state. In this section, we first show how symbolic analysis can be done for a system with synchronous messages. We then show how our methods can be used for systems with asynchronous messages.

3.1 Synchronous Messages

We do symbolic reachability of a system with synchronous messages by constructing a ***reachability graph*** of its configurations. A reachability graph is a directed graph with each node representing a marking, and a directed edge from one marking (say M_1) to another (say M_2) if there is a transition that takes the DPN from marking M_1 to M_2. The edge is labeled with the name of the transition. We allow coordinates of a marking to be symbolic. As an example of symbolic analysis consider the 2-out-of-3 problem.

Figure 2: Example of symbolic analysis of DPN

The 2-out-of-3 problem is a good abstraction of many resource contention problems. Assume that a memory scheduler has three memory blocks and that any process requires two memory blocks to execute. A non-preemptive procedure for such a system with n processes is given in Figure 2. We place n tokens in the state p_1 to signify n processes and three tokens in the state p_5 to signify availability of three memory blocks. To analyze the solution, we draw a reachability graph of its configurations. The initial configuration is $(n, 0, 0, 0, 3, 0)$. With this configuration only a mem transition can take place, resulting in the configuration $(n-1, 1, 0, 0, 2, 1)$ which is explored next. This procedure is continued until all nodes in the graph have been explored. Following it in our example, we find that a deadlock exists if the number of processes is greater than or equal to 3 (see Figure 2).

As in the above example, assume that symbolic reachability graph G is finite. G can also be viewed as a finite state automata. Let $C^{(k)}$ represent interleaving of k clients, and S represent the server. The above system, then can be written as $C^{(k)} \Box S$. Let j be the highest index such that $n-j$ is one of the component in one of the states of the reachability graph. Then,

$$L(G) = L(C^{(k)} \Box S) \qquad \forall k \geq j$$

To see this, consider any $k \geq j$. On substituting for n, the value of k, all the coordinates of states in G will be legal. Now consider any string s. By virtue of the construction of G, s can be simulated in G if and only if it can also be simulated in S and $C^{(k)}$.

Alternatively, we can do asymptotic analysis by replacing n with $*$. Doing a simple reachability indicates that the system can deadlock.

With the brute force method of taking the cross product of all possible states of all processes, there would be 4^{25} states for a system with 25 processes, in contrast to 9 states that need to be explored if the symmetry is exploited. We note here that the saving results from two sources. Firstly, we do not make any distinction between identical processes. Hence, a configuration in which process 1 is in state 1 and process 2 is in state 2 is treated equivalent to the configuration in which the states of both processes are switched. Secondly, if the number of processes is treated as one of the parameters and the size of the symbolic reachability graph is finite, then we have used the fact that the configuration $(n-1,1,0,0,2,1)$ is valid for all $n \geq 1$.

Since the change in the configuration of a DPN is always additive to coordinates, the above construction of the symbolic reachability graph can be carried out automatically. It is sufficient to keep the negative term for each place with a symbolic coordinate. The chief disadvantage of this method is that the reachability graph may not be finite. ω-notation[Karp 68] can be used to make the graph finite but due to the loss of information it can only solve the coverability problem[Peterson 81]. As this method is independent of the issues that arise due to the symbolic nature of coordinates, we do not discuss this method here and refer interested readers to [Peterson 81].

3.2 Asynchronous Messages

We will be interested in the input-output behavior of processes using asynchronous messages. We adapt our model for input-output behavior as follows. We prefix each transition with a $+$ or a $-$. A $+$ before some symbol represents that it is an input (or a receive) event, and a $-$ represents an output (or send) event. Thus, $+a$ represents input a, and $-a$ represents output a. Given any string s of such symbols we can break it into its input and output parts, called $in(s)$ and $out(s)$. For example, if $s = +a + b - a + c - c$, then $in(s) = abc$, and $out(s) = ac$. This definition can be extended to languages in a natural manner. We define a function $\mathcal{B} : 2^{\Sigma^*} \rightarrow 2^{\Sigma^* \times \Sigma^*}$ as follows:
$\mathcal{B}(L) = \{(i, o) | \exists x \in L : i = in(x) \wedge o = out(x)\}$.
This functions takes a language as input and returns a set of input-output pairs for that language (its i/o behavior).

Two different languages may have the same i/o behavior as shown by the following example:
$L_1 = (\epsilon, +b, +b - c, -c)$
$L_2 = (\epsilon, +b, +b - c, -c, -c + b)$
However, i/o behavior for both languages is given by
$\{(\epsilon, \epsilon), (b, \epsilon), (b, c), (\epsilon, c)\}$

Two languages are identified with the same input-output behavior because we lose sequencing relationship between events in input and output channels. In the above example, we know that if the input is b, then output is c. However, we do not know which happens first. In both cases, however, if the entire input i is present at the input channel, then the output o of the process will always be such that (i, o) is in the input-output behavior. This characterization is sufficient for many practical purposes. In particular, input-output characterization is very useful to model the situation in which output of one process becomes input of the other. We formally define input-output semantics of a process on Σ_P as $BehP \subseteq (\Sigma_P)^* \times (\Sigma_P)^*$, with the following constraint:

(i) $(x, y) \in BehP \Rightarrow \forall x' \leq x \;\; \exists y' \leq y : (x', y') \in BehP$.
(ii) $(x, y) \in BehP \Rightarrow \forall y' \leq y \;\; \exists x' \leq x : (x', y') \in BehP$

The constraint (i) simply says that input is considered to affect a system as a stream of events. Thus, if input x produced y, then any prefix of x must also produce a prefix of y. Similarly, (ii) says that output is also a stream of atomic events. If y is produced as a result of x, then every prefix of y must be a possible output of some prefix of x.

We note that we have concerned ourselves only with the finitary behavior of processes. Thus, input-output behavior is described using ***finite*** strings. As a result, this model cannot be used to analyze properties such as fairness which require specification of infinite behavior of processes[Manna 90]. Since many properties of systems can be analyzed using only finite sequences, we do not consider this as a serious limitation of our model.

Some examples of i/o-processes are:
$ZERO_A = (A, \phi)$; process that does nothing
$IDENT_A = (A, \{(x, x) | x \in A^*\})$; process that copies input to the output
$RUN_A = (A, \{(x, y) | x, y \in A^*\})$; process that can do anything
$EPS_A = (A, (\epsilon, \epsilon))$; process that accepts only ϵ
$BUF_A = (A, \{(x, y) | y \leq x\}$; process that copies input to output with delay.

Our main motivation for defining input-output composition is to formalize and exploit the notion of serial composition in which the output of one process becomes input of the other. We define ***serial*** product between two processes denoted by P;Q as

$(x, y) \in BehP; Q$ if and only if $\exists z : (x, z) \in BehP \wedge (z, y) \in BehQ$

It can be easily shown that ; is an associative operation with ZERO as its zero, and IDENT as its identity. We now assume that processes are connected through unbounded reliable FIFO channels. We use P^o as a short form for $P; BUF$. We note that
$(x, y) \in BehP^o$
$= (x, y) \in BehP; BUF$
$= \exists z : (x, z) \in BehP \wedge (z, y) \in BehBUF$
$= \exists z : (x, z) \in BehP \wedge y \leq z$
$= \exists v : (x, yv) \in BehP$
It is the last condition that we will use as the definition of P^o. Considering o as a unary operator, it is easy to see that it is idempotent, i.e., $(P^o)^o = P^o$.

As shown in Figure 3, our system consists of multiple clients operating in parallel. The input-output behavior of two processes is given by the following Lemma. From now on we will treat a process P equivalent to its $BehP$. Thus, instead of writing $(x, y) \in BehP$, we will write $(x, y) \in P$. We first derive an expression for input-output behavior of two

Figure 3: Asynchronous System with Multiple Buffer Processes

processes running asynchronously.

Lemma 1 $P\|Q = \{(x,y) | \exists x_p, x_q, y_p, y_q : (x_p, y_p) \in P, (x_q, y_q) \in Q, x \in x_p \| x_q, y \in y_p \| y_q\}$

Proof:

$\quad (x,y) \in \mathcal{B}(L(P)\|L(Q))$
$=\{$definition B $\}$
$\quad \exists s \in L(P)\|L(Q) : in(s) = x \wedge out(s) = y$
$=\{$definition $\|$ $\}$
$\quad \exists s_p \in L(P), s_q \in L(Q) : s \in (s_p\|s_q) \wedge in(s) = x \wedge out(s) = y$
$= \{$ breaking s $\}$
$\quad \exists s_p \in L(P), s_q \in L(Q), x_p, y_p, x_q, y_q : in(s_p) = x_p \wedge in(s_q) = x_q$
$\quad \wedge\, out(s_p) = y_p \wedge out(s_q) = y_q \wedge x \in x_p\|x_q \wedge y \in y_p\|y_q$
$=\{$definition B $\}$
$\quad \exists x_p, y_p, x_q, y_q : (x_p, y_q) \in P \wedge (x_q, y_q) \in Q \wedge x \in x_p\|x_q \wedge y \in y_p\|y_q$

The above Lemma shows that during interleaving of two processes, an input symbol is executed by either of the process. Similarly, output of both machines are also interleaved.

We next show that an asynchronous execution of two processes with independent buffers is equivalent to their execution with a single buffer.

Lemma 2 $(P\|Q)^o = P^o\|Q^o$

Proof:

$\quad (x,y) \in (P\|Q)^o$
$= \{$ definition o $\}$
$\quad \exists v : (x, yv) \in (P\|Q)$
$= \{$ Lemma 1 $\}$
$\quad \exists v, x_p, x_q, y_p, y_q : (x_p, y_p) \in P \wedge (x_q, y_q) \in Q \wedge (x \in x_p\|x_q) \wedge (yv \in y_p\|y_q)$
$\Rightarrow \{$ property $\|, o\}$
$\quad \exists x_p, x_q, y'_p, y'_q : (x_p, y'_p) \in P^o \wedge (x_q, y'_q) \in Q^o \wedge (x \in x_p\|x_q) \wedge (y \in y'_p\|y'_q)$
$= \{$ Lemma 1 $\}$
$\quad (x,y) \in P^o\|Q^o$
$= \{$ Lemma 1 $\}$

$\exists x_p, x_q, y_p, y_q : (x_p, y_p) \in P^o \wedge (x_q, y_q) \in Q^o \wedge (x \in x_p \| x_q) \wedge (y \in y_p \| y_q)$

$\Rightarrow$ { property of o }

$\exists x_p, x_q, y_p, y_q, v_p, v_q : (x_p, y_p v_p) \in P \wedge (x_q, y_q v_q) \in Q \wedge (x \in x_p \| x_q) \wedge (y \in y_p \| y_q)$

$\Rightarrow$ $\{v \in v_p \| v_q\}$

$\exists x_p, x_q, y_p, y_q, v_p, v_q, v : (x_p, y_p v_p) \in P \wedge (x_q, y_q v_q) \in Q \wedge (x \in x_p \| x_q) \wedge (yv \in y_p v_p \| y_q v_q)$

$=$ { Lemma 1 }

$\exists v : (x, yv) \in (P \| Q)$

$=$ { definition o }

$(x, y) \in (P \| Q)^o$

The above Lemma can be easily generalized to the following Theorem.

Theorem 3 Let $P_0, ..., P_{n-1}$ be any n processes. Then, $(P_0 \| ... \| P_{n-1})^o = P_0^o \| ... \| P_{n-1}^o$

Proof: Using induction on n.

Theorem 3 can be used to reduce the state-space of a system considerably. Instead of analyzing the system with k FIFO channels, it may be sufficient to do so with only one channel. This is illustrated in Figure 3. This theorem may be used to analyze an asynchronous version of the 2-out-of-3 problem with each client having an independent channel to the server.

4 Induction Analysis: Ring Topology

As the number of processes in a ring may be a large, variable or unknown quantity, the construction of the global state graph is not feasible. It is desirable to have a method that analyzes the network with a small number of processes, and then generalize results to the network with a larger number of processes. The key idea that can be frequently applied for ring topology is that of ***induction***. The principle of induction states that if an observer cannot distinguish between two systems with i and $i+1$ processes with any input to both systems, then he also cannot distinguish between a system with i processes and any other system with more than i processes. It follows that it is sufficient to analyze the network with $i+1$ processes for any input-output assertion on more than i processes. We illustrate this by analyzing the dining philosopher's problem. If the number of philosophers is large, a simple reachability technique will run into a state explosion problem. Symbolic techniques are not applicable to this problem because the behavior of each philosopher is not identical as they interact with different entities as opposed to a client-server model in which all servers interact with the same entity - the server. To analyze the problem with a large number of philosophers, we analyze it for a small number of dining philosophers (two in our example). We then show that the same analysis will hold for any number of philosophers. We next describe the problem, a deadlock-free solution and its automatic analysis.

4.1 The Dining Philosophers Problem

This problem, due to Dijkstra, requires an algorithm for philosophers who are sitting on a circular table. There are five philosophers and five forks, each of which is between two philosophers. There is a bowl of spaghetti in the center which can be eaten by any philosopher but its tangled nature requires that a philosopher use both his left and right forks.

A solution to this problem which assumes synchronous communication is as follows. A philosopher when hungry either picks up both the forks simultaneously or waits for them to be available. This way of picking forks guarantees that there will not be any deadlock. To express our solution, we assume that philosopher i ***owns*** fork i and needs to ask only the right neighbor for the use of $i+1^{th}$ fork. For convenience we will use $u_{i,j}$ to denote that $philosopher_i$ picks up $fork_j$ and $d_{i,j}$ to denote that $philosopher_i$ puts down $fork_j$. With this notation, Figure 4 shows the solution expressed in the DPN model.

Figure 4: Dining Philosophers: Analysis

To show that the solution is free from deadlock, we could use a computer to explore the reachable states. Automatic analysis, in the past, meant exploring the cross product of all possible states of five philosophers and five forks (or hundred philosophers and hundred forks for a hundred philosopher problem). Our technique, in contrast, exploits the symmetry in the problem so that the complexity of analysis for five philosophers is the same as that of, say, one hundred philosophers. Various steps in our technique are as follows:

- Let $SYS_k = (PHIL_i \Box .. \Box PHIL_{i+k-1})$. Find the smallest value of k for which $SYS_k = SYS_{k+1}$. For most symmetric cases $k = 1$ or 2 will suffice. For dining philosopher $SYS_1 = SYS_2$ as shown in Figure 4.

- To analyze a ring with any number of units, say n, it is sufficient to analyze it with $k + 1$ units. Thus, for our case it is sufficient to analyze the system with two philosophers to make any assertion about a system with five or one hundred philosophers.
- We next construct a reachability graph for two philosophers and find that there is no state with out-degree equal to zero (see Figure 4). We conclude from this that the system with five philosophers will also be deadlock free.

4.2 Induction Analysis with Filters

Observe that simple induction required that the observer not be able to detect the difference on ***any*** input. This constraint may prove too restrictive to apply induction techniques for certain problems. Therefore, we relax the condition using the concept of ***filters***. Filters capture the condition that not all inputs may be possible for the system, and therefore we are willing to call two systems equivalent as long as their outputs do not differ on possible inputs.

Figure 5: Mutual Exclusion in a Ring

We illustrate the use of filters by a mutual exclusion algorithm in a ring network. Clarke et.al.[Clarke 86] use the same example to illustrate their manual induction technique. Dijkstra[Dijkstra 86] also uses the same example to show how regular expressions can be used to prove the correctness of certain algorithms. His proof, again, is manual. The mutual exclusion problem in a ring of processes is as follows. The machines are connected in a ring fashion and can communicate with their neighbors. Each process can be in one of the three states: normal (n), delayed (d) or critical (c). A process can execute the critical region only if it is in the critical state. The objective is to ensure that at most one machine at a time is in a critical state. We introduce the notion of a token which is held by a single machine. To avoid passing tokens unnecessarily, we introduce a request signal which indicates an interest in the token. A process that wants to execute the critical region, and does not have the token, gets delayed. Following the algorithm, tokens are sent to the left, whereas request signals are sent to the right (see Figure 5). We will also color each of the process as white or black depending upon whether an interest in the token exists to the left. Figure 5 shows the example of a distributed mutual exclusion algorithm in a ring network expressed in the DPN model.

If we try to apply the induction technique that was used for dining philosophers we find that step 1 is not applicable, that is, there does not exist any k for which SYS_k is the same as SYS_{k+1}. This can also be seen intuitively from the algorithm. An observer can detect the number of processes he is connected to by sending multiple token messages. The number of processes in a system would be equal to the maximum number of token messages that are absorbed by the system.

Figure 6: (a) Composition of Two Processes with Filter (b) Filter

To solve this problem, we use the notion of filters to constrain the observer to send at most one more token message than it receives from the output. We now show the steps in the modified induction technique using the mutual ring example.

- Model all the constraints on the input output behavior through a process called FILTER. Figure 6 shows such a filter for our example.

- Verify that a process in the system indeed satisfies the constraint imposed by the filter. If we substitute all request messages in an ENTITY by ϵ, t_{i-1} by t_I and t_i by t_O we do get the filter as a result.

- Find the smallest k such that a filtered system with k units is identical to a filtered system with $k+1$ units. That is,
$FILTSYS_k = (ENTITY_i \Box .... \Box ENTITY_{i+k-1} \Box FILTER)$. For our example we find that $FILTSYS_1 \neq FILTSYS_2$ but $FILTSYS_2 = FILTSYS_3$. It is easy to check that $SYS_k \neq SYS_{k+1}$ for any value of k.

- Thus, from the principle of induction we deduce that it is sufficient to analyze the algorithm with three processes to make an input-output assertion on any number of processes greater than three.

4.3 Asynchronous Messages

In this section, we consider a ring of processes in which process i may receive messages from process $i-1$, and it can send messages only to the process $i+1$. We assume that the link between processes is asynchronous, FIFO and reliable. We use the transition $(q, a/b, r)$ as an abbreviation for two transitions, one after another, representing $(q, +a, q')$ and $(q', -b, r)$. This convention enables us to specify each process as a finite state machine with each transition labeled with input and output.

A process P is called complete iff $(x, y) \in P \Rightarrow \forall x' \geq x \; \exists y' \geq y : (x', y') \in P$. This property may not be true for all processes, but may be desirable if the process is required to respond to all inputs. We also say that $P \leq Q$ if $BehP \subseteq BehQ$.

Our aim is to analyze a ring of processes connected through reliable FIFO links (BUF processes). The following Lemma results in simplification of such networks by combining many channels into one.

Lemma 4 $(P;Q)^o \leq P^o;Q^o$

Proof:
    $(x, y) \in (P;Q)^o$
$=$ { definition of o }
    $\exists v : (x, yv) \in (P;Q)$
$=$ { definition of ; }
    $\exists v \exists z : (x, z) \in P \wedge (z, yv) \in Q$
$=$ { exchange quantification }
    $\exists z \exists v : (x, z) \in P \wedge (z, yv) \in Q$
$\Rightarrow$ { Property of o}
    $\exists z : (x, z) \in P^o \wedge (z, y) \in Q^o$
$=$ { definition of ; }
    $(x, y) \in (P^o;Q^o)$

The inequality in the other direction require Q to be complete.

Lemma 5 If Q is complete, then $P^o; Q^o \leq (P; Q)^o$

Proof:

$(x, y) \in (P^o; Q^o)$
= { definition of ; }
$\exists z : (x, z) \in P^o \wedge (z, y) \in Q^o$
= { definition o}
$\exists z, u, v : (x, zu) \in P \wedge (z, yv) \in Q$
⇒{ Q is complete }
$\exists z, u, v' : (x, zu) \in P \wedge (zu, yv') \in Q$
= { definition of ; }
$\exists v' : (x, yv') \in P; Q$
= { definition o}
$(x, y) \in (P; Q)^o$

To see that the lemma does not hold if Q is not complete, consider the following example:
$P = \{(\epsilon, \epsilon), (a, p), (ab, pq)\}$
$Q = \{(\epsilon, \epsilon), (p, c)\}$
$(P; Q)^o = \{(\epsilon, \epsilon), (a, \epsilon), (a, c)\}$
$P^o; Q^o = \{(\epsilon, \epsilon), (a, \epsilon), (a, c), (ab, \epsilon), (ab, c)\}$

The generalization of the above Lemma for any number of processes requires the following observation.

Lemma 6 If P and Q are complete, then P;Q is complete.

Proof:
Consider $(x, y) \in P$. Let $x' \geq x$
$(x, y) \in P$
= { definition ; }
$\exists z : (x, z) \in P \wedge (z, y) \in Q$
⇒ {P is complete }
$\exists z, z' \geq z : (x', z') \in P \wedge (z, y) \in Q$
⇒ {Q is complete }
$\exists z', y' \geq y : (x', z') \in P \wedge (z', y') \in Q$
= { definition ; }
$\exists y' \geq y : (x', y') \in P; Q$

Lemma 7 Let P_i for $i = 1..n-1$ be complete processes. Then,
$(P_0; P_1; ...; P_{n-1})^o = P_0^o; P_1^o; ...; P_{n-1}^o$

Proof: Using induction on n and Lemmas 4, 5, and 6.

Now, we can analyze any chain of identical processes using the following Theorem. In this section we use P^i to mean $P; P; ...i\ times$.

Theorem 8 Let P be any process such that $P^k = P^{k+1}$ for some k. Assume that $(P^k)^o$ satisfies some input-output constraint. Then so does $(P^o)^n$ for any $n \geq k$.

Proof: Using induction on n. Assume that the theorem is true for $n = m$ which is at least k. Then,

$$\begin{aligned}
& (P^o)^{m+1} \\
= & \{ \text{ Lemma 7 } \} \\
& (P^{m+1})^o \\
= & \{ \text{ definition exponent } \} \\
& (P^{k+1}; P^{m-k})^o \\
= & \{ \text{ Hypothesis } P^k = P^{k+1} \} \\
& (P^k; P^{m-k})^o \\
= & \{ \text{ definition exponent} \} \\
& (P^m)^o \\
= & \{ \text{ Lemma 7 } \} \\
& (P^o)^m
\end{aligned}$$

We can use Theorem 8 to analyze a ring of n processes connected to each other with buffers. From the theorem, we know that the input/output behavior is the same as that of k processes with a buffer at the end.

We illustrate an application of the theorem using a fault-tolerant token ring system[Misra 83]. There are two tokens flowing in the system - green and red. Each process expects to see them alternating one after another. If any of them sees the same colored token twice, it assumes that the other token is lost and generates it. Figure 7(a) shows the state machine representation of the process. We find that $P = P^2$. Therefore, given a chain of any number of processes, we analyze it with just one process followed by a buffer. In this example, we can easily show that the response on any string is correct. For example, the output of the system is always in $(gr)^*$

We now extend our model of the system to incorporate faults. We will assume that a buffer can never lose two consecutive messages. We model unreliable buffer as composition of two processes - ERR and BUF. ERR as shown in the Figure 7(b) models losing of a token. Since composition is associative, we associate ERR with P instead of BUF. Thus, the system can be represented as n number of Q processes connected through reliable asynchronous buffers, where $Q = P; ERR$. Q is shown in Figure 7(c). It can again be seen that $Q = Q; Q$. Therefore, we conclude that it is sufficient to analyze the process Q connected to itself through a buffer. We do this analysis through simple reachability where each node in the reachability tree denotes the state of the process Q and the state of the channel. It can again be checked that the output is always in $(gr)^*$ in spite of any loss of non-consecutive messages.

Another example of a property that we can check is the ***unstability*** of the network, that is, whether the number of messages in the channel can grow in an unbounded manner. Doing the analysis for a single process, we find that the reachability tree terminates and therefore the system is stable (see Figure 7(d)).

We note that most of ring networks exhibit some asymmetry; initially all machines,

Figure 7: Fault-Tolerant Token Ring System

except one, are in the same state. The machine that has different initial state may be in possession of a token, or responsible for initiating computation in the network. Instead of keeping asymmetry in the system, we assume that input channel of the one of the process is non-empty and contains messages such as “token” or “initiate”.

5 Conclusions and Future Work

There is an acute need for systems that can analyze distributed systems. Automatic analysis of even finite state systems runs into the problem of state space explosion. Since most distributed systems show symmetry, we suggest techniques that exploit symmetry to reduce the state space. DPN is a useful model to represent symmetric distributed systems. We use symbolic reachability to analyze systems with a star topology expressed in the DPN model. We also use induction to reduce the number of process that need to be analyzed in a ring network. In this paper, we have shown application of these techniques to many examples. Some other examples are discussed in [Garg 88c].

Many interesting questions arise from this work. In this paper, we saw how machines connected in a star or a ring topology can be analyzed. Some of the other interesting topologies are regular graphs such as hypercubes. An interesting task for investigation is the generalization of these techniques for identical processes connected in such regular topologies. For application of the induction technique, we need to find a k such that the system with k processes is equivalent to a system with $k+1$ processes. It is easy in our examples where k had small values (1 and 2). There needs to be a more general algorithm for selecting k.

6 References

[**Aggarwal 84**] S.Aggarwal, R.P.Kurshan, "Automated Implementation from Formal Specification" *Protocol Specification, Testing, and Verification, IV*, North Holland 1984.

[**Apt 80**] K.Apt, N.Francez, W.de Roever,"A Proof System for Communicating Sequential Processes," *ACM Trans. on Programming Languages and Systems* , Vol. 2,3(July 1980) pp 359-385.

[**Blumer 86**] T.P.Blumer and D.P.Sidhu, "Mechanical Verification of Automatic Implementation of Communication Protocols," *IEEE Trans. on Softw. Engg.*, Vol 12, 8 August 1986, pp 827-843.

[**Chandy 88**] K.M.Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, 1988.

[**Clarke 86a**] E.M.Clarke, E.A.Emerson, A.P.Sistla, "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications", *ACM Trans. on Programming Languages and Systems 8,2* (1986), 244-263.

[**Clarke 86b**] E.M.Clarke, O. Grumberg and M.C.Browne, "Reasoning about Networks with many Identical Finite-State Processes," *Proc. Symposium on Principles of Distributed Computing,* 1986, pp 240-248.

[**Clarke 87**] E.M.Clarke, O. Grumberg, "Avoiding The State Explosion Problem in Temporal Logic Model Checking Algorithms," *Proc. Symposium on Principles of Distributed Computing,* 1987 pp 294-303.

[**Dijkstra 85**] E.W. Dijkstra, "Invariance and Non-determinacy," *Mathematical Logic and Programming Languages,* C.A.R. Hoare and J.C. Shepherdson, Eds. Prentice-Hall, 1985, pp 157-163.

[**Dijkstra 90**] E.W. Dijkstra, C.S.Scholten, *Predicate Calculus and Program Semantics* Springer-Verlag, 1990.

[**Dill 86**] D.L.Dill, E.M.Clarke, "Automatic Verification of Asynchronous Circuits using Temporal Logic ", *IEE Proceedings 133, pt. E* (Sept. 1986), 244-263.

[**Dong 83**] S.T.Dong, "The Modeling, Analysis, and Synthesis of Communication Protocols," Ph.D. Dissertation, University of California at Berkeley, 1983.

[**Garg 88a**] V.K.Garg, "Specification and Analysis of Concurrent Systems Using the STOCS model," *Proc. of Computer Networking Symposium,* Washington D.C. 1988, pp. 192-200.

[**Garg 88b**] V.K.Garg, "Specification and Analysis of Distributed Systems with a Large number of Processes," Ph.D. Dissertation, University of California, Berkeley, 1988.

[**Garg 88c**] V.K. Garg, "Analysis of Distributed Systems with Many Identical Processes", *8th IEEE International Conference on Distributed Computing Systems*pp 358-365, San Jose, California, June 1988.

[**Garg 89**] V. K. Garg, "Modeling of Distributed Systems by Concurrent Regular Expressions", ***Proc. 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols,*** Vancouver, Dec 1989. Also published by North-Holland, 1990, pp. 313-327.

[**Garg 91a**] V. K. Garg, C.V. Ramamoorthy, "ConC: A Language for Concurrent Programming", ***Computer Languages Journal*** Vol. 16, No. 1, January 1991 pp 5-18. a preliminary version appeared in ***IEEE International Conference on Computer Languages,*** Miami, Florida, Oct 1988.

[**Garg 91b**] V. K. Garg, M.T. Raghunath "Concurrent Regular Expressions and their Relationship to Petri Net Languages," ***to appear Theoretical Computer Science Sept. 1991.***

[**Gerhart 80**] S.L.Gerhart, et al., "An Overview of Affirm: A Specification and Verification System," ***Proc. IFIP 80***, pp 343-348, Australia, October 1980.

[**Hoare 85**] C.A.R. Hoare, ***Communicating Sequential Processes,*** Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.

[**Karp 68**] R.Karp, and R.Miller, "Parallel Program Schemata," RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New York (April 1968).

[**Kurshan 85**] R.P.Kurshan, "Modeling Concurrent Processes," ***Proc. of Symposia in Applied Mathematics,*** 1985.

[**Kurshan 89**] R.P.Kurshan, Ken McMillian, "A Structure Induction Theorem for Processes," ***Proc. Eighth Annual ACM Symposium on Principles of Distributed Computing***, Edmonton, Canada, Aug 1989, pp 239-248.

[**Lamport 84**] L.Lamport, F.B. Schneider, "The 'Hoare Logic' of CSP and All That," ***ACM Trans. on Programming Languages and Systems***, 6,2(April 1984).

[**Manna 90**] Z. Manna, A. Pnueli, "A Hierarchy of Temporal Properties," ***Proc. Ninth Annual ACM Symposium on Principles of Distributed Computing***, Quebec, Canada, Aug 1990, pp 377-408.

[**Milner 80**] ***A Calculus of Communicating Systems***, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.

[**Misra 81**] J.Misra, K.M.Chandy, "Proofs of Networks of Processes," ***IEEE Trans. on Softw. Engg.*** SE-7,4(July 1981) pp 417-426.

[**Misra 83**] J.Misra, "Detecting Termination of Distributed Computations Using Markers," ***Proc. of the 2nd ACM Symposium on Principles of Distributed Computing***, Montreal, Aug. 1983, pp. 290-294.

[**Murata 84**] T. Murata, "Modeling and Analysis of Concurrent Systems," in ***Handbook of Software Engineering,*** ed. C.R.Vick and C.V.Ramamoorthy, Publ.Van Nostrand Reinhold, pp 39-63, 1984.

[**Needham 84**] R.M. Needham,A.J.Herbert, "The Cambridge Distributed Computing System," Publ. Addison-Wesley Publishing Company, 1984.

[**Peterson 81**] J. Peterson, *Petri-Net Theory and Modeling of Systems,* Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.

[**Pratt 86**] V. Pratt, "Modeling Concurrency with Partial Orders," *International Journal on Parallel Programming,* Vol. 15, No. 1, Plenum Publishing Corporation, Belgium, Feb 1986, pp 33-71.

[**Reisig 85**] W. Reisig, *Petri Nets, An Introduction,* Lecture notes in Computer Science, Springer-Verlag, 1985.

[**West 78**] C. West, "An Automated Technique of communication protocol validation," *IEEE Trans. on Communications,* vol. 26, no. 8, pp. 1271-1275, August 1978.

[**Zafirolpulo 80**] P. Zafirolpulo, C. West, H. Rudin et al., "Towards analyzing and synthesizing protocols," *IEEE Trans. on Communications,* vol. 28, no. 4, pp. 651-660, April 1980.

7 Appendix A

A concurrent regular expression is any expression consisting of symbols from a finite set Σ and $+$, $.$, $*$, $\Box$, $||$, α, and ϵ with certain constraints as summarized by the following definition.

- Any a that belongs to Σ is a regular expression (r.e.). A special symbol called ϵ is also a regular expression. If A and B are r.e.'s, then so are A.B (concatenation), A+B (or), A^* (Kleene closure).
- A regular expression is also a *unit* expression. If A and B are unit expressions then so are $A||B$ (Interleaving) and A^α (Indefinite Interleaving closure).
- A unit expression is also a concurrent regular expression (cre). If A and B are cre's then so is $A \Box B$ (synchronous composition).

Many examples of systems that can be modeled by concurrent regular expressions are described in [Garg 89, 91a]. We now state the following result without proof.

Theorem 9 [Garg 91b]: The class of languages described by concurrent regular expressions is the same as that described by DPN.