Protocol Verification Using Database Technology

OPHIR FRIEDER, MEMBER, IEEE, AND GARY E. HERMAN, MEMBER, IEEE

Abstract-We **describe a novel application of database technologies in communications networks: *protocol verification* on a *parallel database machine*. We introduce an approach to protocol verification that exploits database algorithms executing on a commercially available, parallel architecture called a hypercube multicomputer. With this approach, we seek to achieve the high degree of computational parallelism necessary to explore rapidly the global-state space of even very complex protocols, significantly reducing the time required to verify a protocol and allowing formal verification to be included as part of the process of protocol design. Our approach is based on the relational database algorithms for a hypercube system presented in [3], [4] and the relational algebra approach to verification of finite-state protocols presented in [17], [18].**

I. Introduction

IN A COMMUNICATIONS network, a protocol is a list of rules/policies that controls and synchronizes the interactions of entities in the network. A difficult practical and theoretical issue in the design of protocols is *protocol verification*, the validation of the logical correctness of the rules governing the interactions of network entities. When protocols are modeled as communicating finite-state machines, *reachability analysis* can be used in the protocol design phase to explore the global states of the system to detect undesirable behaviors, e.g., deadlocks and unreachable states, exhibited by the protocol under design. Once detected, these flaws can be corrected and the new version of the protocol tested again.

While reachability analysis has been used for formal verification of protocols of low to moderate complexity [8], [24], [26], the practical use of reachability analysis for more complex interactions has been constrained by the problem of state space explosion. That is, as the network of communicating finite-state machines increases in complexity, the total number of possible global states grows very rapidly. Searching the global-state space to determine invalid states and detect design errors becomes so time consuming as to be impractical as an integral part of the process of protocol design. One approach to this problem is to investigate design and analysis techniques for protocols that effectively reduce the size of the global-state space that must be actively explored. However, an alternative approach, which forms the basis for our research, is to apply parallel processing techniques to search the global-state space more rapidly. As parallel processing machines incorporating large numbers of processing elements leave the research arena and become commercially available, this latter approach has the potential for significant, practical impact on protocol verification problems.

Manuscript received December 10, 1987; revised September 15, 1988.
The authors are with Bellcore, Morristown, NJ 07960-1910.
IEEE Log Number 8826072.

Use of parallel processing techniques (e.g., [1]) to improve the efficiency of protocol verification requires resolution of several issues commonly encountered in developing efficient parallel algorithms. These include balancing the computational load across many processors, achieving an appropriate balance between communications and processing for the algorithm, and synchronizing algorithm execution across many processing nodes. In this paper we show that, since the process of protocol verification can be described in terms of relational algebra [17], [18], algorithms previously developed for efficient execution of database operations on a parallel machine [3] can be applied directly to reduce dramatically the time required for protocol verification. The algorithms supporting the database operations resolve the issues of balancing and synchronization; further 'algorithm development specific to parallel implementation of protocol verification is not required. Thus, in this paper we describe a novel application of database technologies in communications networks: *protocol verification on a parallel database machine.*

Specifically, we describe an efficient parallel implementation of reachability analysis for protocols described as communicating finite-state machines, based on the original algorithms described in [17], [18] and, using database operations, describe extensions of the basic approach to encompass protocols described in terms of the extended finite-state automata (EFSA) model [9]. The target implementation environment of this approach is a parallel processing machine called a *hypercube multicomputer.* We show that describing the verification process in terms of relational algebra and executing the relational database operations on a hypercube make possible formal verification of protocols with numbers of global states several orders of magnitude larger than protocols considered "difficult" (e.g., the NBS Class IV transport protocol [22]) today. We also show that this approach is scalable; that is, "larger" verification problems can be made tractable by application of larger amounts of computational parallelism, using the same database algorithms.

In this approach, protocols are represented as a set of tables, or *relations,* with each row, or *tuple*, in the relation describing a state and potential transition from that state in the finite-state machine representation of the protocol. Using these relations, the reachable global states can be determined by an iterative sequence of relational join, projection, union, and difference operations that

0733-8716/89/0400-0324$01.OO

eventually generates a global-state transition relation for the system. This final relation can be examined by specific database queries, again described in terms of relational algebra, to detect the presence of undesired behavior. Although the expression of the protocol and the verification process in relational algebra is relatively straightforward, the large volume of data and number of comparisons involved in verifying even a simple protocol make formal verification using conventional database systems quite time consuming in practice. In this paper, we show that general purpose database algorithms executing on a parallel processing machine can significantly reduce the time required to generate the global-state space and examine it for design flaws.

In Section II, we provide an overview of the relational model of protocol verification introduced in [17], [18]. Extensions to this model are also described. Section III describes the hypercube multicomputer and the algorithms for relational database operations designed to execute in that environment. Estimates of the performance improvements achievable by parallel execution of verification algorithms in the proposed system are described in Section IV. A summary of our results is presented in Section V.

II. A Relational Model of Protocol Verification

We provide only a brief overview of the relational algebra approach to protocol verification originally described in [18]; the reader is referred to the original paper for a more complete and formal description. As in [18], we illustrate the approach using the example of the simple connection establishment protocol presented in [26]. The state transition diagrams for this protocol are illustrated in Fig. 1.

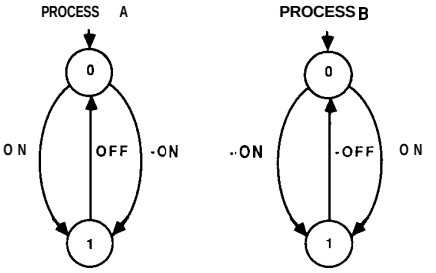


Fig. 1. State transition diagrams for a simple connection-establishment protocol.

The protocol verification process proceeds as a sequence of single-scan (selection, column substitution, column renaming) and *multiscan* (join, projection, difference, and union) database operations. Multiscan operations are so-called because they require that values of attributes in each row (representing a "state" or "state transition" in the protocol sense) be compared to values in every other row in the relation, effectively requiring multiple scans of the data. As we describe in Section IV, execution of the protocol verification algorithm on a parallel machine is heavily dominated by time required to execute the multiscan relational operations. The other, single-scan, operations involved (selection, renaming, and replacement) do not significantly affect the performance of our architecture for "large" protocol verification tasks. Consequently, we emphasize the role of multiscan operations.

Basically, the verification process proceeds in five steps.

Step 1: Transform the digraph protocol specification into tables, called transition relations; a transmission relation ($H\text{-}A$ and H_B) and reception relation ($R\text{-}A$ and R B) exist for processes A and B, respectively. As Table I illustrates, the attributes of each relation include the current state, the event triggering a state transition from that state, and the state to be entered upon encountering the triggering event (a message reception or transmission).

Step 2: Create the system transition relation for a *steady system* [18] — defined to be a system in which only one message is allowed to be transmitted at a time. In a steady system, no message may be sent until the previous message is received, and processes do not receive messages simultaneously. Stable states are global states with both channels empty; transitions between stable global states occur when the triggering event in the reception relation of one process matches the event in the transmission relation of the other process. Thus, the global set Z of possible steady transitions is found by the join of the transmission relation for process A with the reception relation for process B union the join of the reception relation for process A with the transmission relation for process B, in both cases for the join condition TRIGGER-A = -TRIGGER-B. In the notation of [11],

$$Z = (H\_A \text{ TIMES } R\_B \text{ (WHERE TRIGGER-A = -TRIGGER-B)) UNION}$$
$$(H\_B \text{ TIMES } R\text{-}A \text{ (WHERE TRIGGER-B = -TRIGGER-A)).}$$

The relation Z is a relation on the set of attributes { PRESENT-STATE-A PRESENT-STATE-B, TRIGGER-A, TRIGGER-B, FUTURE-STATE-A, FUTURE_STATE_B}. The relation Z is the set of global transitions defined by the protocol state transition rules when only one process may send a message at a time.

All of these possible global steady transitions typically are not reachable from the initial state pair (0, 0 in Fig. 1). We determine the set of reachable states P by beginning with the initial state $P_0 = \langle 0, 0 \rangle$ and searching for reachable global states through an iterative sequence of joins involving I. Essentially this process starts the system at state (0, 0); the first join determines the states P_1 that can be reached directly from (0, 0). The second join

TABLE I
PROCESS TRANSITION RELATIONS FOR THE SIMPLE CONNECTION ESTABLISHMENT PROTOCOL. (a) PROCESS *A* TRANSMISSION RELATION, H-A. (b) PROCESS *A* RECEPTION RELATION, *R-A*. (c) PROCESS *B* TRANSMISSION RELATION, *H-B*. (d) PROCESS *B* RECEPTION RELATION, ***R_B***

Present State A	Trigger A	Future State A
0	-ON	1

(a)

Present State A	Triggger A	Future State A
0	ON	1
1	OFF	0

(b)

Present State B	Trigger B	Future State B
0	-ON	1
1	-OFF	0

(c)

Present State B	Triggger B	Future State B
0	ON	1

(d)

yields the global states P_2 that can be reached directly from P_1; the sequence continues until no new states result. The set of reachable global states P is then given by the union of the P_i relations. With P determined, the set of reachable steady transitions J is given by

$$J = \text{PJOINZ}.$$

The cardinality of J (the number of reachable transitions) is less than or equal to the cardinality of Z (the number of possible legal transitions); as we show in Section IV, typically J is much smaller than I. Table II illustrates the set of reachable steady transitions for our example protocol.

Step 3: In this step, we begin the transformation of the steady transition relation J into the global-state transition relation G. Reference [18] describes this part of the transformation as the result of the union of four separate joins of a relation comprising a single tuple on projections of restrictions of relation J; the joins in this step can be replaced by an equivalent set of more easily computed single scan selection and attribute append operations. This process separates the effects of message transmission by ***A,*** message reception by ***B,*** message transmission by ***B,*** and message reception by ***A,*** on global-state transitions and introduces the concept of channels for messages sent from ***A*** to ***B*** (A-TO-B) and from ***B*** to ***A (B-TO-A).*** The introduction of the concept of channels and unreceived messages in those channels increases the size of the relations involved. For example, if the channel can contain one unreceived message, the cardinality of G following this step is exactly twice that of ***J.*** Each row in ***J*** (a transition resulting from a message transmission-reception pair) generates two rows in G, one for the transition that occurs when the message is sent, and the second for the transition resulting from the corresponding message reception. G, for our example, is shown in Table III under the assumption that a channel can contain only a single message. Fig. 2 interprets G as a finite-state automaton (FSA).

TABLE II
REACHABLE STEADY TRANSITION RELATION, *J*

Present state A	Present State B	Trigger A	Trigger B	Future State A	Future State B
0	0	-ON	ON	1	1
0	0	ON	-ON	1	1
1	1	OFF	-OFF	0	0

Step 4: In this step, we remove the restriction that the system is operated steadily (that is, both processes may transmit messages simultaneously) and expand G to include new global states arising due to multiple messages present in the transmission channels, up to the maximum message capacity N of the channels. This process involves an iterative sequence of at most N joins of the global transition relation G with transmission (H) and reception (R) relations of the two processes. The resulting G is a relation on (4 N + 6) attributes where N is the maximum number of messages allowed to be outstanding in the channel. At this point, the global-state transition relation G includes a tuple for all allowable (under the rules of the protocol) combinations of current states (attributes PRESENT-STATE-A and PRESENT_STATE_***B***), current states of the channel (PRESENT_***B***_TO_***A***[1], . . . , PRESENT_***B***_TO_***A***[N] and PRESENT_***A***_TO_***B***[1], . . . , PRESENT-A-TO ***B***[N]), the message(s) that will trigger the next transition (TRIGGER-A, TRIGGER-B), the states that ***A*** and ***B*** will reach when the next message is transmitted or received (FUTURE-STATE-A, FUTURE-STATE- ***B***) , and the states the channels will reach when the next message is received or transmitted [FUTURE_***B***_TO_***A***[1], . . . , FUTURE_***B***_TO_***A***[N] and FUTURE_***A***_TO_***B***[1], . . . , FUTURE ***A***_TO_***B***[N]). This process also generates a second relation S, the global process state relation, that includes all possible states of the processes and channels. Relation ***S*** is used in testing the protocol for design errors. The final global transition space G for our example is presented in Table IV and as an FSA in Fig. 3.

Step 5: Once the global-state transition relation G has been determined, the protocol can be tested for design errors using relational algebra. ***Deadlock states*** are detected by taking the difference of two projections on G. This operation finds those reachable state pairs ⟨ FUTURE-STATE-A, FUTURE-STATE-B ⟩ that do not also appear as ⟨ PRESENT-STATE-A, PRESENT_STATE-B ⟩ pairs and, therefore, cannot be left once reached. Similarly, projections on S can identify all combinations of process state and incoming messages ***(A,*** PRESENT-B-TO-A ⟩ and ⟨ ***B,*** PRESENT-A-TO-B ⟩ that the protocol allows to occur. If the transition rule for any such pair cannot be found in the process reception

TABLE III
STEADY GLOBAL-STATE TRANSITION RELATION (STEP 3)

Present State A	Present State B	Present B to A	Present A to B	Triggger A	Triggger B	Future State A	Future State B	Future B to A	Future A to B
0	0	0	0	0	-ON	0	1	ON	0
0	0	0	0	-ON	0	1	0	0	ON
0	1	ON	0	ON	0	1	1	0	0
1	0	0	ON	0	ON	1	1	0	0
1	0	OFF	0	OFF	0	0	0	0	0
1	1	0	0	0	-OFF	1	0	OFF	0

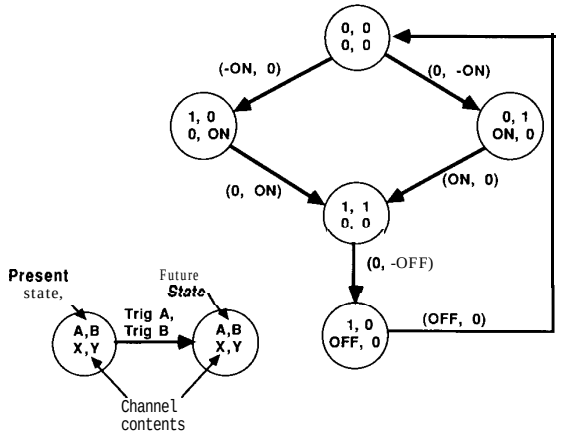


Fig. 2. Steady global-state transition diagram.

TABLE IV
FINAL GLOBAL-STATE TRANSITION RELATION, G

Present State A	Present State B	Present B to A	Present A to B	Trigger A	Trigger B	Future State A	Future State B	Future B to A	Future A to B
0	0	0	0	0	-ON	0	1	ON	0
0	0	0	0	-ON	0	1	0	0	ON
0	1	ON	0	-ON	0	1	1	ON	ON
0	1	ON	0	ON	0	1	1	0	0
1	0	0	ON	0	-ON	1	1	ON	ON
1	0	0	ON	0	ON	1	1	0	0
1	0	OFF	0	OFF	0	0	0	0	0
1	1	0	0	0	-OFF	1	0	OFF	0

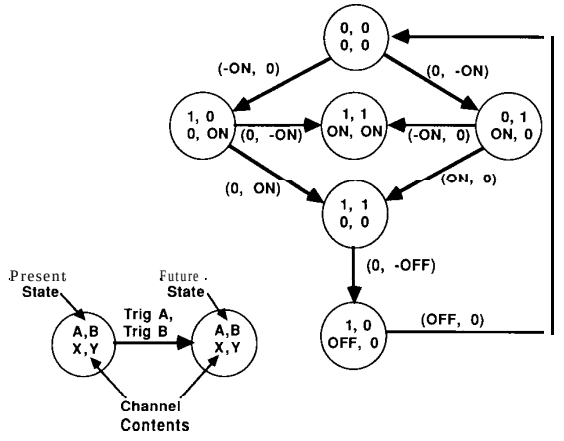


Fig. 3. Final global-state transition diagram.

relations ***R-A*** and ***R-B,*** then the protocol is *incompletely specified. Nonexecutable interactions* are determined by taking the differences between each of the ***H*** and ***R*** relations and projections on G to detect those transition rules in the ***H*** and ***R*** relations that are never exercised by the protocol.

For our example protocol, Step 5 identifies the deadlock state ***(A:*** 1, ***B:*** 1, A-TO-B: ON, A-TO-B: ON $\rangle$, a design flaw in the protocol that occurs when both processes leave state 0 by simultaneously transmitting ON messages.

1) Model Extensions: In addition to verifying protocols that are represented as finite-state automatons (FSA's), the relational algebra verification model can be extended to support the verification of protocols represented in the extended finite-state automaton model (EFSA) [9]. In the EFSA model, each *super-state* for process A consists of a set of internal state variables, $A = \{a_1, a_2, a_3, \cdots, a_p\}$, and corresponds to a set of states in the FSA model. An *instantiation* of a superstate equates a specific value to each internal state variable a_i, $1 \leq i \leq p$. Since each superstate corresponds to a set of states in the FSA model, protocols comprising many states in the FSA model can be described in relatively few states in the EFSA model. Hence, protocols that are too complex to be described in the FSA model can be specified using the EFSA model.

The verification of protocols represented using the EFSA model proceeds in a similar manner to the approach described above. However, instead of, for example, finding the set of valid state transitions through a join with a simple Boolean match operation on the join attribute, a procedure(s) which evaluates the composite set of internal-state variables, a_i, $1 \leq i \leq p$ for process A and b_j, $1 \leq j \leq t$ for process B is invoked. A tuple match (and valid state transition) exists for any tuple (state) pair, one tuple from each of the two joining relations, whose attributes satisfy the transition constraints expressed in the procedure. The individual transition procedure appropriate for a given transition can also be stored as a tuple attribute [23].

Invoking procedures, instead of computing simple joins, significantly increases the computational burden per tuple match. Whereas a join requires only a few CPU instructions per tuple, invoking a procedure may result in the execution of hundreds of instructions per tuple match. Although supporting procedure invocations as part of the verification process complicates the implementation, it also significantly increases the range of applications that can benefit from the proposed verification tool and raises the possibility that his approach may permit formal verification of programs written in languages that can be modeled using an EFSA or other description (cf. [2]). The verification of more complex constructs like programs is beyond the scope of this paper and is left for future work.

To enhance the functionality of the protocol verification system, we can incorporate error backtracking as part of the implementation of the verification system. In verifying complex protocols, it is not sufficient to know that a faulty state can be reached; the faulty path should be identified. To obtain the faulty path, the verification system initially obtains all the unsafe states (see Step 5). Once

these states are identified, a modification of the verification algorithm can be applied iteratively to identify the predecessor states to the known unsafe states. After each step, the newly generated predecessor state tables are output and checked, until, eventually, all possible faulty paths from the initial system state to a faulty state have been identified. At this point, the appropriate corrective modifications to the protocol can be attempted, and the verification process begun once again.

III. Database Machines and Parallel Computer Architectures

Practical use of the approach to protocol verification using algorithms described by relational algebra requires a computing system that implements the multiscan relational operations, like join, very efficiently-much more efficiently than do the general purpose database systems, on which the approach was implemented initially [17]. Many different designs for database machines have been proposed in the past to solve the problem of efficient processing of multiple-join queries. However, due to their specialized nature, database machines have remained mainly in the research arena and have achieved only limited commercial success. The failure of many such special database machines to become commercially viable has stimulated research on the use of more general multiprocessor/computer systems to support database processing.

Several issues must be addressed in designing a parallel architecture and algorithms for efficient execution of database operations. First, the communications overheads and computational loads should be balanced across the nodes in the system, and an appropriate balance should be achieved between communications and computation for the distributed algorithm. Second, means must be provided for synchronizing the actions of the many processing nodes. Third, the architecture and algorithms should be scalable; that is, the architecture and algorithms should allow greater computational parallelism to be usefully applied to solving "larger" problems. Finally, for the approach to be practical, the parallel machine chosen as the execution environment must be available-it must exist.

In the context of protocol verification, a balanced system is achieved through two types of data distribution algorithms. *Load balancing* algorithms evenly distribute the the descriptions of global states (i.e., the tuples in the global-state transition relation G so as to equalize the computational burden across the processing nodes in the system. Even distribution is required because, otherwise, a single node may become overloaded with computational demands and become the system bottleneck. *Compaction* algorithms increase the portion of a relation stored within a given node when necessary to reduce communication overheads for the multiscan operations, like join, that require each tuple in one relation to be evaluated against every tuple in the other relation. Thus, in a parallel environment, one seeks an appropriate balance between the costs of the data distribution algorithms and the resulting benefits in the execution of the database operations themselves.

Performing any task in parallel requires the ability to synchronize the agents which concurrently compute the subtasks. In the database context, prior to computing successor joins, the nodes computing the local joins must be synchronized. In a loosely coupled, distributed environment, operation level synchronization is not easily achieved since there is little or no hardware support for internode coupling. In a parallel machine environment such as a hypercube, node synchronization is directly supported by the architecture of the system and can easily be used by the algorithms implementing the database operations.

Finally, modem parallel architectures are designed to be scalable to very high levels of computational parallelism. In the past, when the cost of the individual hardware components (computers) was relatively high, parallel systems comprising some tens of nodes were investigated. As the individual processing elements became more economical, research addressed systems with hundreds [16], [15], thousands [15], [13], and even tens of thousands of processors [13]. However, the large number of components in a parallel system generates problems of configuring and programming these systems. Our approach to the protocol verification exploits existing software routines executing on a commercially available line of parallel machines that can be configured with a very high degree of parallelism, thus avoiding the need to readdress these problems.

A. The Hypercube Multicomputer

Our implementation environment for the database operations required to perform protocol verification is a message passing architecture called a hypercube. A hypercube is an n-dimensional Boolean cube, Q_n, defined as a cross product of the complete graph K_2 and the $(n-1)$-dimensional Boolean cube Q_{n-1}, with $Q_1 = K_2$. Each node is connected (or adjacent) to each of its $n = \log_2 N$ neighbors where N is the number of nodes. For example, in a four-dimensional cube, Q_4, node 0000 is adjacent to nodes 0001, 0010, 0100, and 1000. Fig. 4(a)-(d) identify one-dimensional (2 node), two-dimensional (4 node), three-dimensional (8 node), and four-dimensional (16 node) cubes, respectively. Note that each system comprises $N = 2^n$ nodes, with n being the cubical dimension of the system. Existing, research-based hypercube machines include CALTECH's Cosmic Cube [21], Jet Propulsion Labs' MARK II [25], and MARK III [12], [19], [6], while commercially available hypercube systems include INTEL's IPSC [16], NCUBE's NCUBE/10 [15], and Floating Point Systems' T/1000 series [13], [20].

Internode communication occurs by sending messages contained in packets. A packet has variable size, with a maximum packet size restricted to, say, 64 Kbytes. A packet can be sent between any two nodes in the system, possibly being routed through intermediate nodes, and is

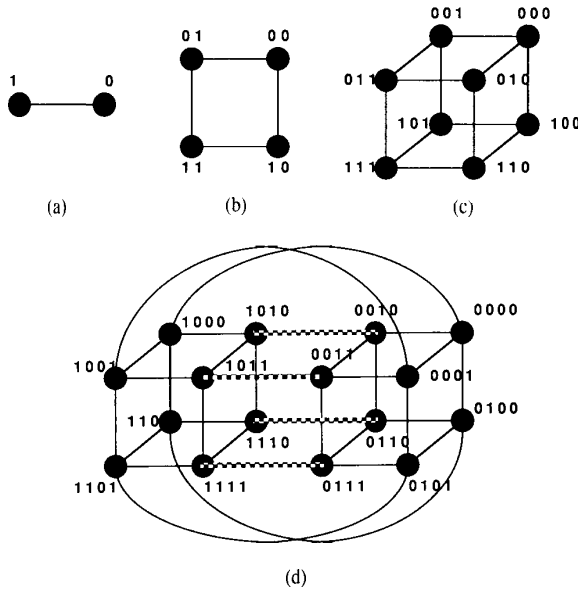


Fig. 4. Various hypercube systems.

used to transfer a "significant amount" of data, in our case tuples, between two nodes. Overhead for packet communications is of two types. Packet transfer overhead is the time required to transmit a packet between nodes; this time is a function of the size of the packet, the internode link speed (ranging from 8 to 64 Mbits/s for existing systems), and the number of links traversed. Packet formation overhead is independent of packet size; typical packet formation times range from 0.5 to 5 ms.

Synchronization among nodes can be achieved either through hardware support or strictly through software. Software synchronization is achieved by forcing a receiving node to "hang" until a message arrives. Thus, the arrival of a message synchronizes the two nodes. This blocking send/receive technique can be generalized so as to enable all nodes within the system to be synchronized. A single designated node globally collects a synchronization message from all nodes in the system which "hang" until an acknowledgment. When the designated node receives a value from all the nodes, it broadcasts a reset value to all the nodes in the system, and all nodes can resume their previous computation. Other hypercube systems synchronize the processors via the use of global synchronization lines. The algorithms we employ assume that hardware global synchronization is available.

B. Relational Database Algorithms on a Hypercube

This section reviews algorithms that implement the relational database operations on a hypercube multicomputer by partitioning the relations, and hence the computational load, across the multiple processors. These algorithms are described in more detail in [3], [5] and are analyzed and evaluated through simulation and actual benchmarks in [4]. To improve computational efficiency, the algorithms presented below have been slightly modified from those presented in [5].

Typically, in implementing parallel algorithms the number of nodes required to achieve best performance in computing the desired task depends on the tradeoff between the average computation expected at each node and the internode communication needed to compute the operation. The optimum system size is the size that provides an approximate timing balance between the computational load assigned to each node and the internode communication which results; that is, in the optimum configuration, the CPU is seldom idle waiting for data to arrive, and communication is seldom blocked waiting for the CPU to complete its operations on local data. If more processors are used, the overhead resulting from the internode communication tends to dominate the computation timing benefits obtained by decomposing the operation into suboperations and computing the suboperations in parallel. In short, as the number of nodes used increases beyond the optimal number, so does the overall operation completion time. This optimum configuration is operation dependent; further, the data must be carefully organized to take full advantage of the multiple processors for a given operation.

The hypercube database primitives are of two types: those that are used to support *dynamic* data redistribution, i.e., the "on-the-fly" reorganization of data to promote a better balanced workload, and those that directly implement the relational operations, such as select, join, etc. We initially provide a brief narrative as well as pseudocode description of the data redistribution primitives. After the explanation of the base primitives, we describe the actual relational database operations-select, project, and join-based on these primitives. Additional details on the redistribution primitives and the database algorithms on the hypercube are available in [14].

1) Data Redistribution: The following redistribution primitives are necessary for the implementation of the relational database operations on a hypercube.

a) Tuple Balancing-redistributes the tuples to achieve a roughly even distribution across all the nodes, avoiding uneven processor execution time. In the tuple balancing pseudocode below, two relations are balanced simultaneously.

1) The local tuple counts of the relations R_1 and R_2 are computed.

2) During each stage j ($1 \le j \le n$, $n = \log_2 N$), the nodes whose addresses differ in thejth bit exchange their local R_1 tuple count via the fast packet transfer primitive, and the node with the greater number of tuples (if any) sends the excess tuples to the other. Simultaneously, the nodes whose addresses differ in the (($j + 1$) mod n)th bit balance R_2. Thus, after completion of this step, the nodes whose address differs in thejth bit contain roughly the same number of R_1 tuples, while the nodes whose address differs in the ((j + 1) mod n)th bit contain roughly the same number of R_2 tuples.

3) The local tuple counts for R_1 and for R_2 are updated at each node.

4) Steps 1 to 3 are repeated n times.

b) Relation Compaction and Replication (RCR)—replicates the smaller relation R_1, originally stored in a cube of dimension n, in such a manner that it will be replicated in each of the two, equal-sized, dimension $n-1$, logical, cube partitions of the original cube. The goal of this primitive is to increase the number of tuples from R_1 stored at each node until one packet size of R_1 is present at each node, or until R_1 has been fully replicated at each node. This primitive ensures that packets used in the join phase will be as fill as possible, and that the packet formation overhead per tuple will be minimized for the cycling primitive.

1) The local tuple count of the relation to compact and replicate is computed.

2) During each stage j ($1 \leq j \leq n$), the nodes whose addresses differ in the jth bit exchange their local R_1 tuple count. RCR is possible if the sum of the R_1 tuple storage volumes in a node pair is less than one full packet size. If RCR is not possible then a global control line is set. If RCR is possible then all nodes transmit their tuples to their paired neighbor.

3) The tuple count for the compacted and replicated relation is updated.

4) Steps 1 to 3 are repeated until either n RCR steps have occurred or a global line indicating the termination of the RCR operation has been set.

c) Relation Compaction (RC)-same as RCR but only one of the logical cube partitions contains the data. This operation compacts the data from a relation into fewer nodes, until a fill packet size of data exists at every node. Again, the goal is to minimize per-tuple packet formation overheads for the cycling primitive.

d) Cycle-create a hamiltonean cycle/ring within each logical cube partition generated by either the RC or the RCR primitives, and pipeline the data packets throughout the cycle/ring. A hamiltonean cycle can be dynamically generated via the use of reflexive Gray codes.

2) Selection: Each node performs local selection in parallel. If the results are to be collected, then an output collection step is incorporated; otherwise no global operation is necessary. As with all the operations, operation termination is signaled via the use of *global synchronization lines.*

3) Projection: Initially, a local projection *(removing nonrelevant columns from each tuple and eliminating local duplicates)* is performed. Since not all nodes will necessarily remove the same number of tuples, i.e., have the same number of duplicates, the tuple distribution across nodes may become skewed as a result of the local projection. In this case, tuple balancing is performed. This is followed by an RC step in which nodes eliminate duplicates between the local tuples and the newly obtained packet. When this step terminates, tuple balancing is performed, and RC is retried. If RC is not possible, then the algorithm enters the cycling phase in which the global duplicate elimination is performed.

4) Join: The join algorithm comprises three basic primitives. First, tuple balancing is performed to ensure even distribution of input tuples. Second, the RCR operation is performed both to reduce the cube (subcube) size, if necessary, and to replicate the smaller relation to enhance parallel processing. Third, the cycling primitive is used to send the tuples of the smaller relation around in a ring and perform local joins in each node.

IV. Performance Evaluation

In this section, we investigate the performance of our proposed protocol verification system. Obviously, an exact analysis of algorithm performance depends critically on the details of the protocol to be verified, particularly the cardinality of the *Z, J,* and G relations computed in the verification process. Because we wish to develop performance estimates that generalize beyond individual protocols, we first examine the growth in state space for several example protocols that can be verified easily on a conventional uniprocessor database system. To provide a framework in which to examine the ability of parallel processing techniques to improve performance in verifying much larger protocols, we then define a synthetic protocol called the *binary tree protocol.* We use this protocol to evaluate the performance of our approach on a protocol with over five million global-state transitions.

A. Examples of State Space Growth

Using a modified version of the REPROVER verification software described in [17], we verified four relatively small protocols: the simple connection establishment protocol described in Section II, the example process interaction protocol from [18], [27], X.21 [26], and the binary synchronous (bisynch) protocol [10]. Table V illustrates the cardinality of the *Z, J,* and G relations for each protocol; the largest, the binary synchronous protocol, has 354 states in its global-state transition relation G. Table V shows a consistent pattern across the four protocols studied. In each, the number of possible transitions Z is large, but the number of transitions reachable from the initial state is only a small subset of those in I. In each case, the size of the global-state transition space $|G|$ is substantially larger than $J|$, roughly equivalent to $|Z|$. Fig. 5 shows the size of the final global-state transition relation G for each protocol plotted as a function of the size $H + R|$ of the original process specification for that protocol. In our sample of four protocols, $|G$ grows slightly worse than linearly with $|H + RI$.

B. The Binary Tree Protocol

Our interest in applying computational parallelism to the problem of protocol verification is to allow the verification of very large protocols, much larger than the relatively simple protocols that we verified using REPROVER. To provide a concrete basis for exploring the potential performance of our approach, we define a simple

TABLE V
RELATION SIZES FOR FOUR EXAMPLE PROTOCOLS

	|H| + |R|	|I|	|J|	|G|
Connection Establishment	3	3	3	8
Process Interaction	7	9	5	22
X.21	51	383	59	238
Bisynch	69	1600	58	354

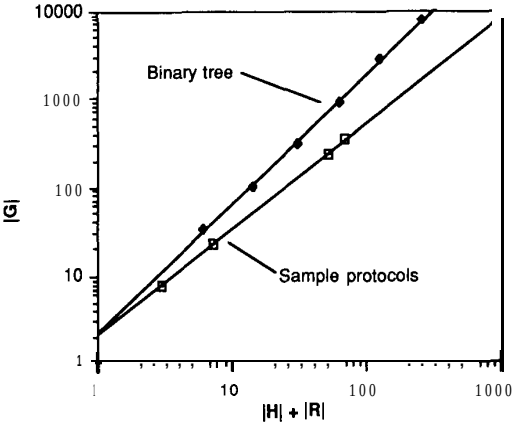


Fig. 5. Size of global-state transition relation G versus number of initial process transitions $H + R$.

family of synthetic protocols called binary tree protocols. As Fig. 6 illustrates, the digraph for each member of this family consists of a binary tree of depth d. A protocol process of depth d comprises $2^d - 1$ states; each of the first $2^{d-1} - 1$ states has one parent state and two child states. One child state is reached by transmission of the message M ($-M$ in our notation). The other child state is reached when message M is received. The final 2^{d-1} states, those at level d in the tree, have only one child state-the initial state of the protocol. At this level, either transmission or reception of message M returns the process to its initial state. The entire message vocabulary of the protocol consists of the single message, M.

While the binary tree protocol is logically trivial, it allows creation of protocols with predictable large state spaces that are purely a function of depth d. The number of tuples in each of *H-A*, *H_B*, *R-A*, and *R_B* relations is $2^d - 1$. Because every state has a transition defined for the only message that can be sent, the Z relation generated in Step 2 has the maximum possible cardinality, $2(2^d - 1)^2$. However, the binary tree protocol has a very restrictive structure; the *J* relation has only $2(2^d - 1)$ tuples. The final global-state transition relation G, with the channel capacity restricted to one message,[1] includes $G(d)$ tuples where

$$G(d) = 3G(d-1) + C(d-1),$$
$$C(d) = C(d-1) - 2^d$$
$$\text{and } C(1) = G(1) = 8.$$

[1]In its simplest form, the binary tree protocol is unbounded in N, the number of unreceived messages allowed in the channel.

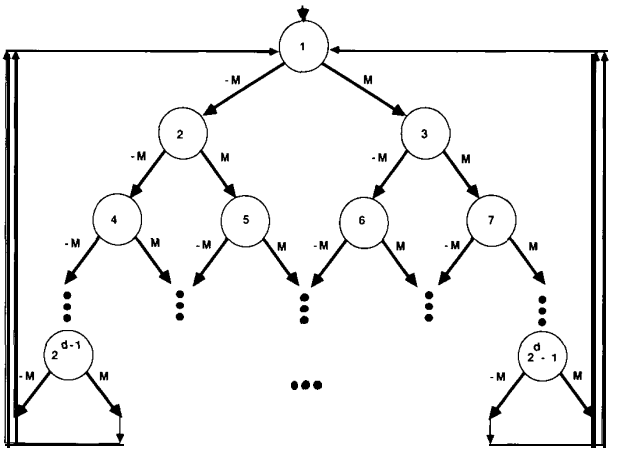


Fig. 6. Binary tree protocol of depth d.

Fig. 5 also shows G versus $H + R$ for the binary tree protocol family for several values of depth d; the state space for the binary tree protocol appears to expand somewhat more rapidly than we observed in our four sample protocols.

A binary tree protocol of depth 7 has 127 transmission and reception rules for each process that permit 32 258 possible system transitions, of which only 254 are reachable. The system has 7540 reachable global-state transitions. REPROVER required over 15 h of CPU time on a SUN 3/160 to execute the verification algorithm for this protocol. In comparison, REPROVER verified the bisynch protocol in 25 min.

C. Verification Performance on a Hypercube

A precise performance model for verification on a hypercube requires a detailed analysis of algorithm timing for each step described in Section II, as well as a model of query compilation, query distribution, data loading, etc., on the hypercube. Our goal in this section is to estimate the relative benefit we can expect in applying parallelism to the problem of protocol verification. To address this question, it is sufficient to evaluate the performance of the hypercube algorithms on the multiscan join operation and neglect the times associated with the remaining single-scan operations such as selection and renaming. Ignoring the times corresponding to the single-scan operations simplifies the analysis and does not significantly effect the results since the join times greatly dominate the single-scan operation times [14].

Throughout this analysis, we assume that each hypercube node consists of a conventional 2 MIPS CPU, independent communications processors with associated buffers, 64 Mbit/s bidirectional communication links, and a large local memory. Besides the link speed, factors affecting internode communication overhead include the delays incurred in packet setup and decomposition and the maximum packet size allowed in the system. Based on existing systems, we assume that the hypercube supports

a maximum packet size of 64 Kbytes and a packet processing time of 3 ms. We assume that each tuple requires 40 bytes of storage; in terms of CPU time, in our evaluation, each tuple comparison in the join operation requires 10 instructions. To explore the value of parallelism in verifying complex protocols, we demonstrate our approach using the depth 10 and depth 13 synthetic protocols described above.

Recall that in the binary tree protocol the number of tuples in each of the *H-A, H-B, R-A,* and *R-B* relations is $2^d - 1$ where d is the depth of the tree. If $d = 10$, Step 1 of the algorithm generates the above four relations, each comprising 1023 tuples. We assume that the tables are loaded into the hypercube system from a "user," and the four tables are all resident at a single node. We begin our analysis with the first stage of Step 2, the generation of all possible transitions Z allowed by the transmission and reception transition relations.

For the depth 10 synthetic protocol, the steady transition relation Z can be computed through a 2046 by 2046 tuple join, combining the *H-A* by *R_B* and *H-B* by *R-A* joins into a single operation. The time necessary to compute this join is shown in Fig. 7. As shown, the minimum execution time of roughly 34 ms. occurs on a two node system; depending on the number of nodes involved, execution time varies from 34 to 70 ms. Because the number of tuples is small and were assumed to exist initially in a single node (the worst case distribution), the balancing and RCR primitives dominate the total execution time, representing 44 and 32 percent, respectively, of the total time for $N = 2$. Note that when the number of processors exceeds 2, the added parallelism degrades the overall performance. The increase in the join processing time results from partitioning the computational load over too large a set of nodes, so that the overhead of internode communication is greater than the computation time reduction obtained via parallel execution.

The result of the 2046 by 2046 tuple join is an output relation Z containing 2 093 058 tuples. We assume that Z is partitioned across the nodes of the hypercube such that no node contains a portion of Z that is more than 4 times the portion contained in any of its neighbors. Determining the the set of actual reachable steady transitions, relation *J* requires a sequence of *d* joins, the most complex of which consists of 2 093 058 by 2046 tuples. Upon completion of Step 2, we obtain a *J* relation comprising 2046 tuples. Transforming *J* into the global-state transition relation G requires a sequence of single and multiscan operations, the most complex of which requires a join of $G \mid$ by $(H + R \mid)$ tuples, or 198 872 by 2046 tuples for the depth 10 binary tree.

Fig. 8 illustrates the effect of increased parallelism on join computation performance in creating the *Z, J,* and G relations. Clearly, computing the largest join (for *J)* dominates the other operations in terms of time required. In forming relation *J,* parallelism is successfully exploited for all hypercube sizes investigated; near linear

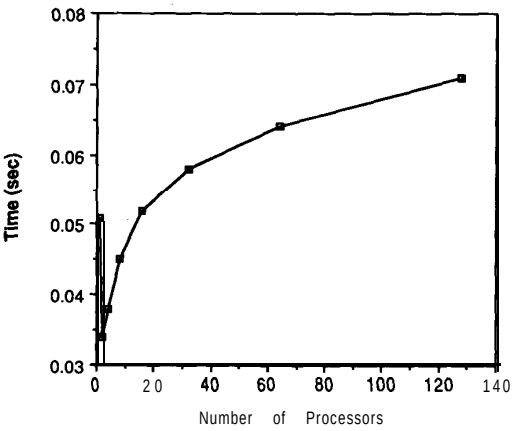


Fig. 7. Join computation times for relation I for depth 10 binary tree protocol.

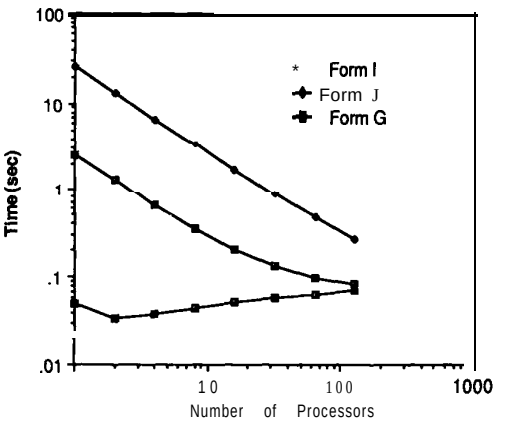


Fig. 8. Join computation times for relations *I, J, G* for depth 10 binary tree protocol.

speed-up is obtained for all hypercube sizes up to 64 nodes, and use of 128 nodes provides a factor of 100 reduction in processing time. In forming relation G, performance fails to improve for systems greater than 32 nodes in size.

Parallelism has even greater impact on more complex protocols. For example, each process in a depth 13 binary tree protocol consists of 8191 initial states and 16 382 transitions. In computing the Z relation, two 8191 by 8191 tuple joins are required. Relation *J* is formed via a sequence of 13 joins, the most complex of which requires the join of two relations of 134 184 962 tuples and 16 382 tuples; the most complex join required to compute G consists of 5 330 788 by 16 382 tuples. The times required to compute these joins are shown in Fig. 9. Linear speed up is now achieved for both the *J* and G join computations. Executing these algorithms on a 128 node hypercube instead of a uniprocessor can reduce computation time from several hours to a few minutes.

Finally, we conclude our analysis by comparing the gains due to parallelism as the depth of the binary tree protocol varies from 2 to 13, or from 32 to 5 330 788

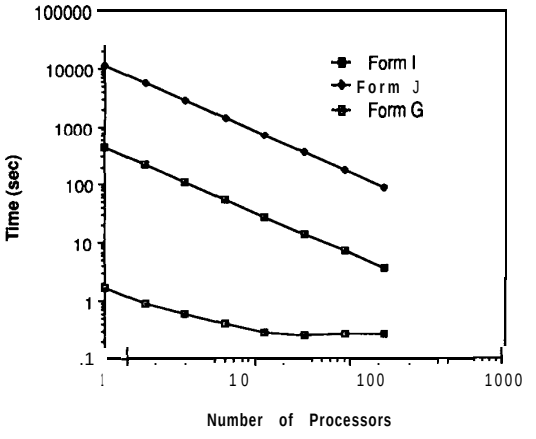


Fig. 9. Join computation times for relations *I, J,* G for depth 13 binary tree protocol.

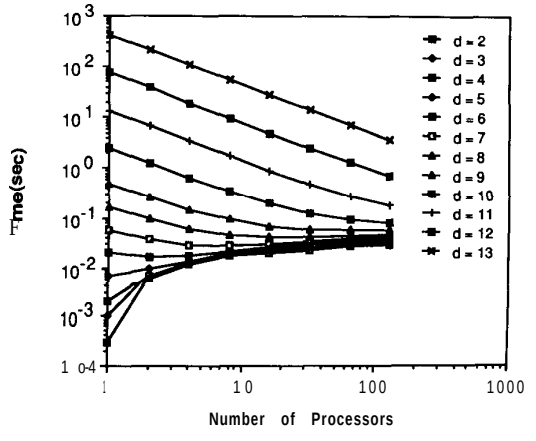


Fig. 10. Join computation times for the global-state transition relation G for the binary tree protocol family.

global-state transitions. Fig. 10 shows the times required to compute the largest join in the process of transforming J into G for the family of binary tree protocols. As shown, as the complexity of the join operation increases (the depth of the protocol is increased), the benefits due to parallelism also increase. For relatively small protocols, parallelism is detrimental.

V. Summary

The verification of complex protocols is an important problem for both the research and the development communities. This paper described an approach to verifying complex communications protocols based on implementing the relational database algorithms for protocol verification from [17], [18] using database primitives designed for the hypercube multicomputer [5]. Suggested extensions to the basic approach included the verification of protocols represented in the EFSA model and the use of backtracking to determine faulty paths that generate undesirable global states. Using a synthetic protocol whose state space growth characteristics resemble those of real protocols, we showed that our approach can achieve near-linear speedup of the verification process for up to 128 processors, or more, depending upon the complexity of the particular protocol involved. Commercial systems come with up to 16 384 nodes. Our results suggest that, when implemented, this approach can reduce the verification time for complex protocols from many hours to a few minutes.

We are implementing the system described here and extending the verification model to a more general class of communicating concurrent processes.

Acknowledgment

The authors would like to thank M. Lai and T. Lee for assistance in this research, M. Agron for her investigations of the behavior of protocols using REPROVER, and C. H. Chow and G. Gopal for their comments on earlier drafts of this paper.

References

[1] S. Aggarwal, R. Alonso, and C. Courcoubetis, "Distributed reachability analysis for protocol verification environments," in Discrete *Event Systems: Models and Applications*, P. Varaiya and A. Kuzhanski, Eds. New York: Springer-Verlag, 1987, Lect. Notes. Contr. Inform. Sci., pp. 40–56.

[2] S. Aggarwal, D. Barbara, and K. Z. Meth, "A software environment for the specification and analysis of problems of coordination and concurrency," *IEEE Trans. Software Eng.*, vol. 14. pp. 280–290. Mar. *1988.*

[3] C. K. Baru and 0. Frieder, "Implementing relational database operations in a cube-connected multicomputer," in *Proc. IEEE Third Int. Conf. Data Eng.,* Feb. 1987, pp. 36-43.

[4] C. K. Baru, 0. Frieder, D. Kandlur, and M. Segal, "Join on a cube: Analysis, simulation, and implementation," in *Proc. 5th Int. Workshop Database Mach.,* Japan, 1987, pp. 74-87.

[5] C. K. Baru and 0. Frieder, "Database operations in a cube-connected multicomputer system," *IEEE Trans. Comput.*, to be published.

[6] B. Beckman, "Distributed simulation and the time warp operating system," *Proc. ACM S.O.S.P.*, pp. 77-93, 1987.

[7] D. Bergmark, J. M. Francioni, B. K. Helminen, and D. A. Poplawski, "On the performance of the FPS T-series hypercube," presented at *Proc. 2nd Conf. Hypercube Multiprocess.*, Sept., 1986.

[8] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *J. ACM, vol. 30,* no. 2. pp. 323-342, 1983.

[9] T. Y. Choi, "Formal techniques for the specification, verification. and construction of communication protocols," *IEEE Commun. Mug.*. ***vol. 23,*** pp. 46-52, Oct. 1985.

[10] C. Chow. M. G. Gouda, and S. S. Lam, "A discipline for constructing multiphase communication protocols," *IEEE Trans. Software Eng.*, vol. 14. pp. 327-338. Mar. 1988.

I I] C. J. Date, *An Introduction to Database Systems, Volume I.* Reading, MA: Addison-Wesley, 1986.

12] G. Fox, "Use of the Caltech hypercube," *IEEE Trans. Software Eng.*, vol. SE-I 1, p. 73, July 1985.

13] K. A. Frenkel, "Evaluating two massively parallel machines," *Commun. ACM, vol. 29,* no. 8, pp. 752-758, 1986.

14] 0. Frieder. "Database processing on a cube-connected multicomputer," Ph.D. dissertation, Univ. of Michigan, Ann Arbor. 1987.

15] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a hypercube supercomputer," in *Proc. Int. Conf. Parallel Process.,* Aug. 1986, pp. 653-660.

[16] Intel iPSC Data Sheet, Order 280101-001, 1985.

[17] M. Y. Lai and T. T. Lee, "Protocol verification using relational da tabase systems," in *Proc. IEEE 3rd Int. Conf. Dutu Eng.*, Feb., 1987, pp. 347-354.

[18] T. T. Lee and M. Y. Lai, "A relational algebraic approach to protocol verification." *IEEE Trans. Software Eng., vol.,* pp. 184–193, Feb. 1988.

[19] J. C. Peterson, J. 0. Tuazon, D. Lieberman, and M. Pniel, "The MARK III hypercube-ensemble concurrent computer," in *Proc. Int. Conf. Parallel Process.*, Aug. 1985, pp. 71-73.

[20] A. P. Reeves and D. Bergmark, "Parallel Pascal and the FPS hypercube supercomputer," in *Proc. Int. Conf. Parallel Process.*, Aug., 1987, pp. 385-388.

[21] C. Seitz, "The Cosmic Cube," Commun. ACM, vol. 28, no. 1, pp. 22-33, Jan. 1985.

[22] D. P. Sidhu and T. P. Blumer, "Verification of NBS class 4 transport protocol," *IEEE* Trans. *Commun.*, vol. COM-34, pp. 781-789, Aug. 1986.

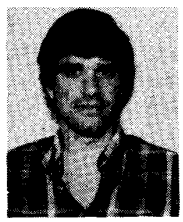
[23] M. Stonebraker, J. Anton, and E. Hanson, "Extending a database system with procedures," *ACM Trans. Database Syst., vol.* 12, no. 6, pp. 350-376, 1987.

[24] C. A. Sunshine, "Survey of protocol definition and verification techniques," in *Computer Networks. New* York: North-Holland, 1978, pp. 346-350.

[25] J. Tuazon, J. Peterson, M. Pniel, and D. Lieberman, "CALTECH/JPL MARK II hypercube concurrent computer," in *Proc. Int. Conf. Parallel Process.*, pp. *666-673,* Aug. 1985.

[26] C. W. West and P. Zafiropulo, "Automated validation of a communications protocol: The CCITT X.2 1 recommendations," *IBM J. Res. Develop., vol. 22, no. 1,* pp. 60-71, 1978.

[27] P. Zafiropulo, C. H. West, D. D. Cowan, and D. Brand, "Towards analyzing and synthesizing protocols," *IEEE Trans. Commun.,* vol. COM-28, Apr. 1980.



Ophir Frieder *(M'87)* received the B.Sc. degree (1984) in computer and communications science and the M.Sc. (1985) and Ph.D. degrees (1987) in computer science and engineering, all from the University of Michigan.

He is currently a Member of Technical Staff at Bell Communications Research. His research interests include parallel and distributed architectures, operating systems, fault-tolerant systems, and database systems.

Gary E. Herman (M'87) received both the B.S.E. (1971) and Ph.D. (1975) degrees in electrical engineering from Duke University, Durham, NC.

From 1976 to 1983 he worked at Bell Telephone Laboratories. In 1984 he was a founding member of the Applied Research Area of Bellcore. He currently is Division Manager of the Network Systems Research Division at Bellcore, where his research interests include several aspects of large scale distributed systems, including scalable transaction system architectures, system verification and testing, and software fault tolerance.