SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 22(3). 245–264 (March 1992)

A Parallel Database-driven Protocol Verification System Prototype

O. FRIEDER
Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, Virginia 22030-4444, U.S.A.

SUMMARY

Protocol verification systems test the specification of communication protocols. A prototype of a protocol verification system based on parallel database primitives for a hypercube multicomputer is described. Using the described prototype, two conventional protocols are verified. To evaluate the performance of verification systems, a synthetic protocol that is to be used as a benchmark is proposed. Extensions to and properties of the synthetic protocol are described. Using this protocol, the performance of the developed verification system is evaluated.

KEY WORDS Protocol verification systems Multiproccssor database Hypercube multicomputer Protocol benchmarks

INTRODUCTION

Protocol verification systems [1–9] test the specification of interprocess communication software, namely *protocols*, and determine their correctness. A protocol is viewed as correct with respect to a set of properties. For example, a protocol is considered correct with respect to *deadlock-freedom* if there does not exist a sequence of transitions that results in deadlock. General background in this topic can be found in the tutorials on protocol verification techniques by Choi, [10] Miller [11] and Yuang, [12] and the text by Holzmann. [13]

Since a communicating process pair may be in any one of many global states, generating and testing each of these states, especially for complex protocols, requires great computational power. Thus, for verification systems to be of practical interest, namely maintaining a low response time, suitable computational resources must be available. We concentrate on parallel protocol verification systems like those described in References 1, 5 and 6, and employ parallelism as a means by which to meet the computational demands of protocol verification systems. We differ from Aggrawal, *et al.* [1] in that they propose a set of loosely-coupled workstations connected in a local area network (LAN) to parallelize their system, whereas our system is

0038–0644/92/030245–20$10.00

Received 1 October 1990
Revised 31 October 1991

based on a (tightly-coupled) hypercube multicomputer. * Aggrawal *et al*. concluded that a maximum of tens of workstations can be used in their implementation while maintaining performance. The analysis by Frieder and Herman, [6] however, demonstrated the effectiveness of using a hypercube multicomputer of up to 128 processors for the verification of highly complex protocols. We focus on verification systems that are built on top of readily available commercial hardware. This excludes the implementation described in reference 5, which is based on a special-purchase system.

The protocol verification system evaluated here is based on the database-driven verification algorithm discussed by Lee and Lai, [7] and on an extension of the parallel relational database effort on a hypercube described by Baru and Frieder. [14] An analytical study of our system, a portable, parallel, database-driven, protocol verification system, was reported by Frieder and Herman. [6] In this paper we review, extend and evaluate a prototype implementation of the system proposed therein.

As there is currently no standard by which to evaluate the performance of multiprocessor verification systems, we propose a synthetic benchmark protocol, and use it to evaluate our system. For generality of use by others, the structure of the proposed protocol is automatically generated and can be represented in multiple forms, e.g. expressions of various types for systems like those in References 1, 2 and 5 and finite state representations for systems like those in References 3, 4, 6, 7 and 8. As the standard for representing protocols is based on the finite state model, we present our description of the benchmark protocol in a finite state format. By incorporating the maximal fanout from any state and the maximal path-length of a typical protocol to be verified by the system, the complexity of generic protocols is captured by the synthetic structure.

We analyse the structure of the proposed synthetic benchmark protocol mathematically in terms of the number of generated global reachable states. As memory and processor capacity limit the capability of a verification system, determining the maximal protocol complexity that the system can verify is crucial. Since protocol complexity can be approximated by the total number of reachable global states, the ability to pre-compute the number of states can reduce the number of the required experimental tests.

Finally, we demonstrate our system by verifying conventional protocols such as X21 [15] and Modified Bi-Synch [3] and evaluate the performance of our system using the benchmark protocol.

The remainder of this paper is organized as follows. We initially present the overall system architecture of our prototype. The multicomputer (hypercube) database primitives used in the implementation are then presented. We continue by illustrating our database-driven protocol verification algorithm that is a modification of the verification algorithm presented in Reference 7. Finally, the structure of the proposed synthetic benchmark protocol is described and an evaluation of a running prototype using conventional protocols and various configurations of the benchmark protocol is presented.

* Although there arc many commonly accepted definitions for multicomputer systems, throughout this paper it is assumed that a multicomputer system is my architecture comprising multiple nodes with each node being a complete computer. That is, each node has a set of resources, namely dedicatcd I/O, memory, and CPU, and is under the independent control of its own operating system. However, it is not neccssary for each computer to be fully configured or have all the resources that are theoretically available under its operating system.

SYSTEM ARCHITECTURE

As developing software for a parallel environment is particularly difficult, especially if the code development is further hindered by the particularities of a given parallel engine, we implemented our protocol verification system using a set of macros developed at Argonne National Laboratories. These macros, described in Reference 16, provide a fully-portable, simulated, distributed-memory environment that hides most of the machine specifics from the development process. If a shared-memory system is desired, communication among processors takes a small number of memory access times, independent of the actual inter-node communication delay. The package employed maintained its own logical clock. It must be noted, however, that the use of these macros introduces significant overheads. Hence, our measurements should be used only to evaluate the potential parallelism of this approach and not as absolute values. A production version of this verification approach would implement the software directly on an actual hypercube with the database primitives incorporated into the operating system.

Control architecture

Each parallel entity is a lightweight process. Processes are logically either masters or slaves; the master process acting as a supervisor with the slaves actually performing the work. Communication between processes is accomplished via the use of dedicated channels. In the environment provided by the Argonne macros, an application is a single master process and a set of slave processes. Processes are statically allocated to processors. If each slave process is allocated one per physical processor, the macro package actually emulates, and not just simulates, the user-defined interconnection network architecture.

The system executes as follows. Initially, the master creates a set of slaves and establishes the master/slave and slave/slave dedicated communications channels. As our protocol verification system is logically based on a hypercube, the inter-slave communication is limited to a nearest-neighbour hypercube topology. In addition, each node directly communicates with the master through a dedicated link. In a production version of this approach, the host machine would act as the master controller.

The master program consists of a set of commands that are issued sequentially to the slaves. All slaves contain the same code and execute only when instructed by the master. In this implementation, the verification algorithm resides at the master process, and the slaves possess the database routines. The actual database processing is performed completely by the slaves and is independent of the protocol verification algorithm. Replacing the verification algorithm in the master process with any other relational sequence of instructions is possible and results in a new database program.

After establishing the channel, the master partitions the verification algorithm into steps. A step is a sequence of database operators. Each step is a command message sent by the master to all the slaves. Each slave executes the command on its local partition of the database, communicating with other slaves when dictated by the database algorithms. Slaves acknowledge command execution termination and await the next command. This continues for all steps in the verification algorithm. Upon completion of the verification algorithm, the master issues a termination command to all the slaves and the program terminates.

Operation logging is performed by the master. Each issued command is recorded. In debugging, this logging mechanism proved to be of a substantial benefit. Under normal execution, the logging of events enables the starting and stopping of the verification algorithm depending on the availability of the system and its resources. For example, if the verification algorithm takes longer than expected, the database already created can be repartitioned across a larger cube. The repartitioning can be done by invoking the balancing operation on a larger cube dimension (see the Section on 'Data redistribution primitives').

Database structure

The entire database is stored as a tree of memory/secondary storage blocks. Blocks are of various sizes and are not-necessarily in contiguous storage. The root of the tree is called database. Relations comprise of one or more blocks, each block containing some number of tuples. All tuples within a relation are of fixed size. In the system, relations are *horizontally partitioned* across the slaves. That is, every slave contains a possibly empty subset of the tuples of each relation *R*. Empty subsets result when the stored relation is small in comparison to the number of slaves. All slaves maintain an identical structure of their portion of the entire database.

Each relation is represented by a relation descriptor block (RDB) that consists of six fields. The Next-RDB field is a pointer to the next relation in the database. Relation-Name is a string identifier that is unique throughout the entire database and designates the relation name. The number of attributes and the total size of each tuple in the relation are stored in the Number-of-Attributes and the Tuple-Size fields, respectively. Both fields are integers. The remaining two fields in the RDB are pointers. The first is a pointer to the first attribute description block (ADB), one ADB per each attribute in the relation. The second pointer points to the first in a list of data description blocks (DDB).

Each ADB also has six fields including a pointer to the next ADB within the relation and a string designating the attribute name. All attributes within a relation must have unique names. To simplify the data packetization and transfer among nodes while still supporting machine portability, four additional integer fields are included in each ADB. These fields correspond to the attribute number within the relation, its corresponding size and offset from the beginning of the tuple (both in bytes), and the domain type of the attribute. Possible domain types include string, short integer, short real, integer, real, and boolean.

Data description blocks have four fields, starting with a pointer to the next DDB. The three remaining fields include an integer indicating the maximum size of the data block in bytes, the number of tuples already stored in the data block, and a pointer to the actual data block. The number of bytes in a data block indicates the physical memory limitation of the data block and is always a multiple of the size of the relation's tuple. The data block itself consists of packed relational data stored in row major form. All memory blocks are dynamically acquired from the operating system.

RELATIONAL DATABASE PRIMITIVESON HYPERCUBES

The primitives used here are of two types: those supporting *dynamic* data redistribution, i.e. the 'on-the-fly' reorganization of data, and those that directly implement the relational operations, such as select, join, etc. For further details, see the tutorial by Frieder. [17]

Data redistribution primitives

Several data redistribution primitives are needed in the implementation of the relational database operations on a hypercube. These primitives are described below. All descriptions assume an N-node system.

Tuple Balancing redistributes the tuples to achieve a roughly even distribution across all the nodes, avoiding uneven processor execution time. The pseudocode for simultaneously balancing two relations is as follows:

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 the j th bit exchange their local R_1 tuple count. 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 the j th bit contain roughly the same number of R_1 tuples, whereas 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.

Relation compaction and replication (RCR) replicates the smaller relation R,, originally stored in a cube of dimension n, in such a manner that it will be replicated in each of the two, equal-sized, $n - 1$ dimension, logical, cube partitions of the original cube. The goal of this primitive is to increase the number of tuples from R, 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 full as possible, and that the packet formation overhead per tuple will be minimized for the cycling primitive. The process implementaiton is as follows:

1. The local tuple count of R, *is* computed.
2. During each stage j ($1 \le j \le n$, $n \log_2 N$), the nodes whose addresses differ in the j th 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 possible then all nodes transmit their tuples to their paired neighbor.
3. The tuple count for the compacted and replicated relations is updated.
4. Steps 1 to 3 are repeated until either n RCR steps have occurred or the termination of the RCR operation has been signalled.

Cycle creates a Hamiltonian ring within each logical cube partition generated by the RCR primitives and pipelines the data packets throughout the ring. A Hamiltonian cycle can be dynamically generated via the use of reflexive Gray codes.

Data repartition by value (DRV) redistributes data according to attribute(s) values.

Each node is assigned a non-overlapping subset of the global attribute domain. The union of the attribute value range assignment across all nodes is the global domain.

1. During each stage j ($1 \leq j \leq n$, $n = \log_2 N$), all nodes partition their local tuples into two sets: those tuples that according to their respective attribute(s) values the final destination are nodes whose j th bit equals 1 and those nodes whose j th bit equals 0.
2. All nodes keep the tuple set whose destination equals the value of their respective j th bit. The other tuple set is sent to the node whose address differs ones own address in only the j th bit. The received tuple set is merged with the kept tuple set.
3. Steps 1 and 2 are repeated n times.

Database primitives

The database primitives are performed as follows.

Selection

As each node has direct access to a dedicated secondary storage device and relations are horizontally partitioned across these devices, our implementation of the selection operation exploits conventional data parallelism. That is, each node performs local selection on its resident data in parallel. If the results are to be collected, then an output collection step is incorporated; otherwise no global operation is necessary.

Projection

Projection is performed in three steps. Initially, attribute trimming (the removal of non-relevant columns/attributes from each tuple) is performed. Secondly, the DRV operation partitions the tuples across the processors according to the values of the attributes that remained after trimming. Finally, local duplicate elimination based on sorting is performed.

Join

Two join algorithms are applied. The first, broadcast based, is employed in cases where it is known that one relation is significantly larger than the other.

The broadcast join comprises of three basic primitives. First, tuple balancing is performed to ensure an even distribution of input tuples. Secondly, the RCR operation replicates the smaller relation, enhancing the available parallelism while reducing the packet formation overhead per tuple in the cycling phase. Thirdly, the cycling primitive pipelines the tuples of the smaller relation, in a ring-like manner, within each local cube partition formed in the second step. Local joins are performed at each node.

The second join algorithm is bucket based and proceeds as follows. Both relations are repartitioned according to attribute values via the DRV primitive. Once partitioned, all nodes locally sort their data and compute the local join.

Bucket based approaches reduce the computations time at the expense of additional inter-node communication. The reduction in computation time results in bucket joins having lower total join processing times, as compared to broadcast-based joins, when both joining relations are comparable in size. [17] The exact disparity in size required for the broadcast join to yield lower processing times than the bucket join is system dependent. That is, the time required to format the packet, the effective internode communication link speed, the routeing architecture employed, the number of nodes in the system, and the type of CPU employed, all play a significant role in determining the required size.

A DATABASE-DRIVEN PROTOCOL VERIFICATION ALGORITHM

The protocol verification algorithm described in Reference 7 and used as a basis for the control structure here is based on the relational database model. Our version incorporates various query optimizations (operation execution reordering) that reduce the size of the intermediate relations and hence the total processing time. As the optimization techniques used do not alter the skeleton/behaviour of the relational verification algorithm presented in Reference 7, we do not discuss them further. Throughout the remainder of this paper, we assume the reader is familiar with the fundamentals of the relational database model. For a comprehensive text on the relation database. see Reference 18.

A protocol verification algorithm

The input to our system is a set of four relations representing the send and receive transitions of each process in the process pair, respectively. The verification algorithm consists of a sequence of relational operations which compute a final global relation. Each tuple in the global relation designates a possible global system configuration. Erroneous states, e.g. those transitions leading to deadlock and unspecified reception, are detected by querying the database.

The code segments provided below illustrate the verification algorithm as implemented in the prototype. Function calls not directly related to the verification algorithm, for example load partitioning routines that maintain a balanced workload across the multiple processor, are not included. It should be noted that the verification algorithm initially proposed by Lee and Lai [7] was modified in the implementation both to improve efficiency and to aid in the parallelization process. The log_fp file descriptor is used in the logging of the operation execution. The verification algorithm can be viewed conceptually as comprising of five basic steps:

1. Convert the finite state description of the protocol into tabular (relational) form. The relations specifying the protocol, namely HA (send transitions of A), RA (reception transitions of A), HB (send transitions of B), and RB (reception transitions of B), are provided as input to the verification system.
2. Create the *steady system* transition relation—defined as a system in which only one message is allowed to be transmitted at a time. [7] The relational database operation sequence required to generate the steady system transition relation, *J*, from the input relations is shown below. As seen, the implementation employs procedures based on the relational database whenever possible (see Figure 1).

```
/* having the selected slaves read in the initial data */

/*IAB=HA  JOIN  RB*/
join(proc_ids, "IAB", "HA JOIN RR WHERE [HA.AM = mi(-1 ) RB.BM ]
   [ A0 = HA.AS B0) = RB.BS AM = HA.AM RM = RB.BM Al = HA.AN B1 = RB.BN ]", log_fp);

/*IBA=RA  JOIN  HB*/
join(proc_ids, "IBA", "RA JOIN HB WHERE [ RA.AM = mi(-1 ) HB.BM ]
   [A0=RA.AS B0=HB.BS AM = RA.AM BM = HB.BM Al = RA.ANB1 = HB.BN log_fp);

/*I=IAB U IBA*/
 union(proc_ids, "I", "IAB UNION IRA", log_fp);

/* R = PROJECT I */
 project(proc_ids, "R", "PROJECT I [ A0 = I.A0 B0 = I.B0 Al = I.A1 B1 = I.B1 ]", log_fp);

flag = TRUE
i = 0;
 while (global_continue && flag)
{
   i ++;
   join(proc_ids, "S1","R JOIN ST WHERE [ (ST.SA = R.A0 ) && (ST.SB = R.B0 ) ]
              [ SA = R.AI SB = R.Bl ]", log_fp):
   join(proc_ids, "TEMP", "SI JOIN SR WHERE [ ( SI.SA = SR.SA ) && ( SI.SB = SR.SB ) ]
              [ SA = SI.SA SB = SI.SB ] ", log_fp);
   diff(proc_ids, "TEMP2", "SI - TEMP", log_fp);
   rename(proc_ids, "SI", "TEMP2", log_fp);
   copy(proc_ids, "ST", "COPY SI", log_fp);
   if (slcount(proc_ids, "ST", log_fp) > 0)
      slappend(proc_ids, "SR", "APPEND ST F", log_fp);
   else
      flag = FALSE;
 }/* endwhile */

/* RS = SR */
 copy(proc_ids, "RS", "COPY SR", log_fp);

/* TAB= IAB JOIN RS */
join(proc_ids, "TAB", "IAB JOIN RS WHERE [ ( IAB.A0 = RS.SA ) && ( IAB.B0= RS.SB ) ]
   [A0=IAB.A0B0= IAB.B0 AM = IAB.AM BM = IAB.BM Al = IAB.A1 B1 =IAB.B1 ]", log_fp);

/* TBA = IBA JOIN RS */
join(proc_ids, "TBA", "IBA JOIN RS WHERE [ ( IBA.A0 = RS.SA ) && ( IBA.B0 = RS.SB ) ]
   [ A0 = IBA.A0 B0 = IBA.BO AM = IBA.AM BM = IBA.BM Al = IBA.A1 B1 = IBA.B1 ]", log_fp);

/*J=TAB U TBA*/
 union(proc_ids, "J", "TAB UNION TBA", log_fp);
```

Figure 1. The generation of a steady system

3. Transform the steady system relation, *J*, into a synchronous global transition relation, *T*. In a synchronous global system, multiple messages may be transmitted simultaneously, however the process transmissions are synchronized. At this stage, relation *T* represents all possible transitions in a synchronous system (see Figure 2).
4. Compute the final global transition relation, *F*. The final relation enumerates

```
/* T0 = TAB JOIN ZER0 (no predicate) */
join(proc_ids "T0", "TAB JOIN ZERO [ A0 = TAB.A0 B0 . TAB.B0 X0 = ZERO.Z YO = ZERO.Z AM =
   TAB.AM BM = ZERO.ZA1 = TAB.A1 B1 = TAB.B1 Xl = ZERO.Z Y1 =TAB.BM ]", log_fp);

/* TEMP = TAB JOIN ZERO (no predicate) */
join(proc_ids, "TEMP", "TAB JOIN ZERO [ A0 = TAB.A1 B0 = TAB.B0 X0 = ZERO.Z YO = TAB.
   BM AM = ZERO.Z BM = TAB.BM Al = TAB.A1 B1 = TAR.B1 Xl = ZERO.Z Y1 = ZERO.Z ]", log_fp);

/* TO = TO + TEMP */
 slappend(proc_ids, "T0", "APPEND TEMP T", log_fp);

/* TEMP = TBA JOIN ZERO (no predicate) */

join(proc_ids, "TEMP", "TBA JOIN ZERO [ A0. TBA.A0 B0 = TBA.B0 X0 = ZERO.Z YO = ZERO.
   Z AM = ZERO.Z BM = TBA.BM Al = TBA.A0 R1 = TBA.B1 Xl = TBA.AM Y1 = ZERO.Z ]", log_fp);

/* TO = TO +TEMP */
 slappend(proc_ids, "T0" ,"APPEND TEMP T",log_fp);

/* TEMP = TBA JOIN ZERO (no predicate) */
join(proc_ids, "TEMP", "TBA JOIN ZERO [ A0 = TBA.A0 B0 = TBA.Bl X0 = TBA.AM YO = ZERO.
   ZAM =TBA.AM BM = ZERO.Z Al = TBA.A1 B1 = TBA.B1 X1 = ZERO.Z Y1 = ZERO.Z ]", log_fp);

/*TO = TO + TEMP */
 slappend(proc_ids, "T0", "APPEND TEMP T", log_fp);

/* TEM = PROJECT T0 */
 project(proc_ids, "TEM", "PROJECT T0 [ A0 = T0.A0 B0 = T0.B0 X0 = T0.X0 Y0 = T0.Y0 ]", log_fp;

/* TEMP = PROJECT T0 */
 project(proc_ids, "TEMP","PROJECT T0 [ A0 = T0.A1 B0 = T0.R1 X0 = T0.X1 Y0 = T0.Y1 ]", log_fp);

/* TEM = TEM + TEMP (delete TEMP) */
 slappend(proc_ids, "TEM", "APPEND TEMP T", log_fp);
```

Figure 2. The generation of a synchronous global transition relation

all possible system configurations. Intuitively, each pass of the while-loop above, computes all the new system configurations that can be reached when either or both communicating processes simultaneously send and/or receive a single message. The sending and/or receiving can occur when the global system is in any of the already known configurations (see Figure 3).

5. Query the database, namely relation *F*, for erroneous states. As all possible transitions are represented as tuples in *F*, using database queries, it is possible to detect the existence of erroneous protocol properties. In Reference 7, queries necessary to detect deadlock, incomplete specification, nonexecutable interaction, etc., are formulated. The code segment above illustrates the query detecting various deadlock conditions (with and without empty channels) (see Figure 4).

```
/*STEM=GO=Gl=TEM*/
 copy(proc_ids, "STEM", "COPY TEM". log_fp));
 copy(proc_ids, "GO", "COPY TEM", log_fp);
 copy(proc_ids, "G1", "COPY TEM". log_fp);

flag = TRUE
 while (global_continue && flag)
{
   /*Tl  =Gl  JOIN  HA*/
   join(proc_ids, "Tl", "Gl JOIN HA WHERE [ ( HA.AS = GI.A0 ) && ( G1.Y0 = cs(0) ) ]
              [ A0= G1.A0 B0 = GI.B0 X0 = G1.X0 Y0 = G1.Y0 AM = HA.AM BM = cs(0)
              Al = HA.AN B1 = G1.B0 X1 = G1.X0 Y1 = mi(-1) HA.AM ]", log_fp);

   /* T2 = G1 JOIN RA */
   join(proc_ids, "T2", "Gl JOIN RA WHERE [ ( RA.AM = G1.X0 ) && ( RA.AS = G1.A0 ) ]
              [A0=G1.A0  B0=G1.FWX0=G1.  X0  Y0=GI.Y0AM  =  RA.AMBM  =cs(0)
              Al = RA.AN B1 = GI.B0 Xl = cs(0)) Y1 = G1.Y0 ]", log_fp);

   /*Tl=Tl+T2*/
   slappend(proc_ids, "Tl", "APPEND  T2  T", log_fp);

   /*T2=Gl  JOIN  HB  */
   join(proc_ids, "T2", "Gl JOIN HB WHERE [ ( HB.BS = GI.B0 ) && ( G1.X0 =  cs(0) ) ]
              [A0=G1.A0B0=G1.B0X0=CI.X0Y0=G1.Y0AM=cs(0)BM=HB.BM
              Al =G1.A0BI = HB.BN Xl = mi(-1) HB.BM Y1 = G1.Y0 ]", log_fp);

   /* T1 = T1 + T2 (destroy T2) */
   slappend(proc_ids,  "T1","APPEND  T2  T",  log_fp):

   /*T2=Gl  JOIN  RB*/
   join(proc_ids, "T2", "Gl JOIN RB WHERE [ ( RB.BM = GI.Y0 ) && ( RB.BS = G1.B0 ) ]
              [A0=G1.A0B0=Gl.B0X0=G1.X0Y0=G1.Y0AM=cs(0)BM                =RB.BM
              Al = G1.A0 B1 = RB.BN Xl = G1.X0 Y1 = cs(0) ]", log_fp);

   /* T1 = T1 + T2 (destroy T2) */
   slappend(proc_ids, "T1", "APPEND T2 T", log_fp);

   /*Tl=T1-T0*/
   diff(proc_ids, "T", "T1 - T0", log_fp);
   rename(proc_ids, "T1", "T", log_fp);

   /*T0=T0+Tl*/
   slappend(proc_ids, "TO", "APPEND T1 F", log_fp);

   /* G1 = PROJECT T1 */
   project(proc_ids, "G1","PROJECT T1 [ A0 = T1.A1 B0 = TI.B1 X0 = T1.X1 Y0 = T1.Y1 ]". log-fp);

   /*Gl=G1-G0*/
   diff(proc_ids, "T", "G1- G0", log_fp);
   rename(proc_ids, "G1", "T", log_fp);

   if (slcount(proc_ids, "Gl", log_fp) > 0)
      slappend(proc_ids, "GO", "APPEND G1 F". log_fp);
   else
     flag = FALSE;

 }/* endwhile */
```

Figure 3. The generation of the final global transition relation

```
    /**********************/
  /*deadlock state detection */
    /**********************/

  /* DL= PROJECT T0 */
  project(proc_ids, "DL", "PROJECT T0 [ A0 = T0.A 1 B0 = T0.B1 X0 = T0.X1 Y0 = T0.Y1 ]", log_fp);

  /* TEMP = DL JOIN T0 */
  join(proc_ids, "TEMP","DL JOIN T0 WHERE [ ( DL.A0 = T0.A0 ) && ( ( DL.B0 = T0.B0 ) &&
     ( (DL.X0 = T0.X0 ) && ( DL.Y0 = T0.Y0 ) ) ) ] [ A0 = DL.A0 B0 = DL.X0 X0 = DL.X0 Y0 = DL.Y0 ]",
log_fp);
   balance(proc_ids, "TEMP", log_fp);

  /* DL = DL - TEMP */
   diff(proc_ids, "TEMP2", "DL - TEMP", log_fp);
   rename(proc_ids, "DL", "TEMP2", log_fp);

     /***************************************/
   /* deadlock state with no message in the channel */
     /*****************************************/

  /* DS = SELECT DL */
  select(proc_ids, "DS", "SELECT DL WHERE [ ( DL.X0 = cs(0) ) && ( DL.Y0 = cs(0) ) ]", log_fp);
```

Figure 4. Querying the protocol for deadlock

K-process protocol verification: database structures

The protocol verification algorithm described in the preceding section verifies the specification of a two process protocol. An extension of the described relational database-driven verification algorithm that supports the verification of protocols comprising *k* processes is provided in Reference 19. Here we briefly comment on the modifications to the underlying database system necessary to support the verification of *k* -process protocols.

The described algorithm requires the specification of tables representing the reception and transmission of each process (HA, HB, RA, RB). An equivalent representation maintains only two tables indicating the source, destination, and message type associated with the transmission or reception, respectively.

The use of the two table representation scheme directly supports the verification of k-process protocols. In the reception table, all attributes are always atomic. In the transmission table, the source attribute is always atomic; however, the destination attribute is atomic only when multicast transmission is not supported. (If atomic attributes are required, conventional relational normalization techniques can be employed. The normalization may yield up to 2^k-1 target attributes depending on the degree of multicast supported.)

A BENCHMARK PROTOCOL

As no standard exists for comparing the performance of protocol verification systems, similar to the Gibson instruction mix [20] used in evaluating the performance of CPUs and the Wisconsin benchmark [21] that developed a generic set of relations and associated queries for the evaluation of the performance of database architectures, we

developed a generic protocol as a benchmark for protocol verification systems. We focus on two critical constraints in defining the proposed benchmark *protocol family** structure. First, all benchmarks must be portable. That is, the benchmark must conform to the input requirements of many of the verification systems to be evaluated. Since protocols are represented differently in each system, simple, automated generation of an instance of a benchmark protocol is required. Secondly, to evaluate parallel verification systems, a benchmark protocol must be scalable, i.e. supporting a structured increase in complexity. For a comprehensive treatment of benchmarking techniques, see Reference 22.

The proposed synthetic protocol consists of two identical processes, *A* and *B*, whose representation are the graphs *G(A)* and *G(B)*, respectively. Let *N(G)* represent the nodes of G and E(G) represent the edges of G. Processes *A* and *B* each consists of set of states *S* and transitions *T*, such that $S_A = \{ a_i \mid a_i \in N(G(A))\}$, $S_B = \{ b_i \mid b_i \in N(G(B))\}$, $T_A = \{(a_i \mid a_i) \mid (a_i, a_j) \in E(G(A))\}$ and $T_B = \{(b_i, b_j) \mid (b_i, b_j) \in E(G(B))\}$. It is assumed that the processes are connected by a full duplex, error-free, FIFO channel. No assumptions governing the amount of time a message can remain in the common channel nor the amount of time a process can remain at a given state are made.

Analytically, each process in the benchmark protocol is a complete *m*-ary tree† of depth *d*, with the leaf nodes adjacent to the root‡. Thus, each process consists of

$$\sum_{i=0}^{d} m^i = \frac{1-m^{d+1}}{1-m}$$

states and

$$\sum_{i=1}^{d+1} m^i = \frac{m-m^{d+2}}{1-m}$$

transitions.

The general structure of each process in the protocol family structure is shown in Figures 5(a) and 5(b). Figure 5(a) illustrates the labels for each branch (transition) in the protocol. By convention, the positive arcs represent message receptions and the negative arcs, message transmissions. Since each node can send and receive each message type, the outdegree of every state is even. Hence, *m/2* is an integer. Figure 5(b) designates the state numbering scheme. As shown, state *i* is the parent of *m* states numbered *mi*, *mi*+ 1, *mi+2*, ... (*i* +l) *m* 2, (*i* +l) *m* -1. Every leaf node has *m* transitions to the root. Each node, with the exception of the root, has an indegree of 1 and an outdegree of *m*. The root has an indegree of m^{d+1}. By definition the root node is state 1. As shown, the state labelling is non-continuous, but unique. The numbering scheme presented is a direct extension of the breadth-first node labeling of binary trees. The complexity of the protocol is modified by varying the depth (*d*) and number of message types *(m)*. Figures 6(a)-(c) illustrate (*d* = 1, *m* =4), (*d* =2, *m* =2) and (*d* =2, *m*= 4) protocols, respectively.

---

* We choose the term protocol family since we are actually proposing a set protocols as our benchmark. Each set member is derived from a common structure, but with different generating parameters.

† Although not an actual tree (leaf to root transitions), due to its structural similarity to a tree. we will refer to each process as a tree.

‡ By definition, the root of a tree is of depth *d* = 0,

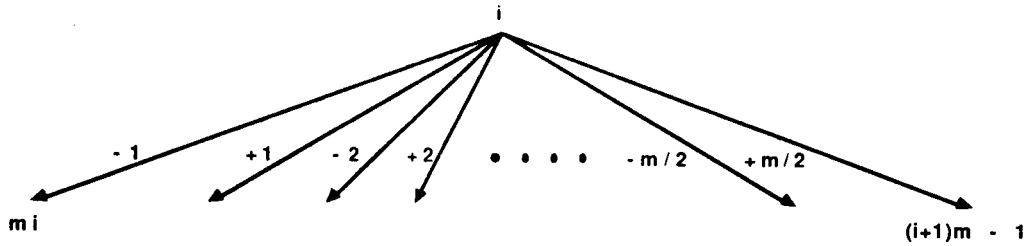

Figure 5. (a) Synthetic protocol message labelling

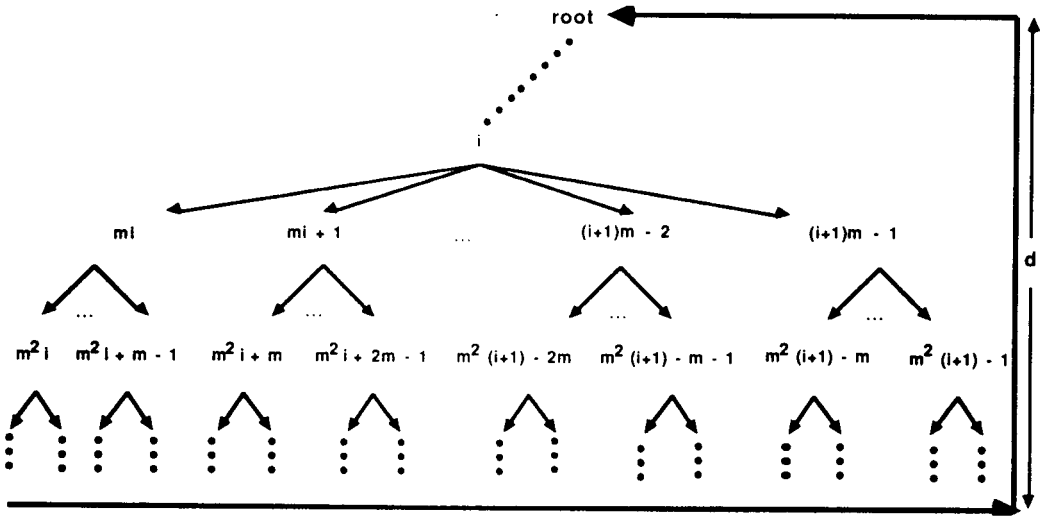

Figure 5. (b) Synthetic protocol state labelling

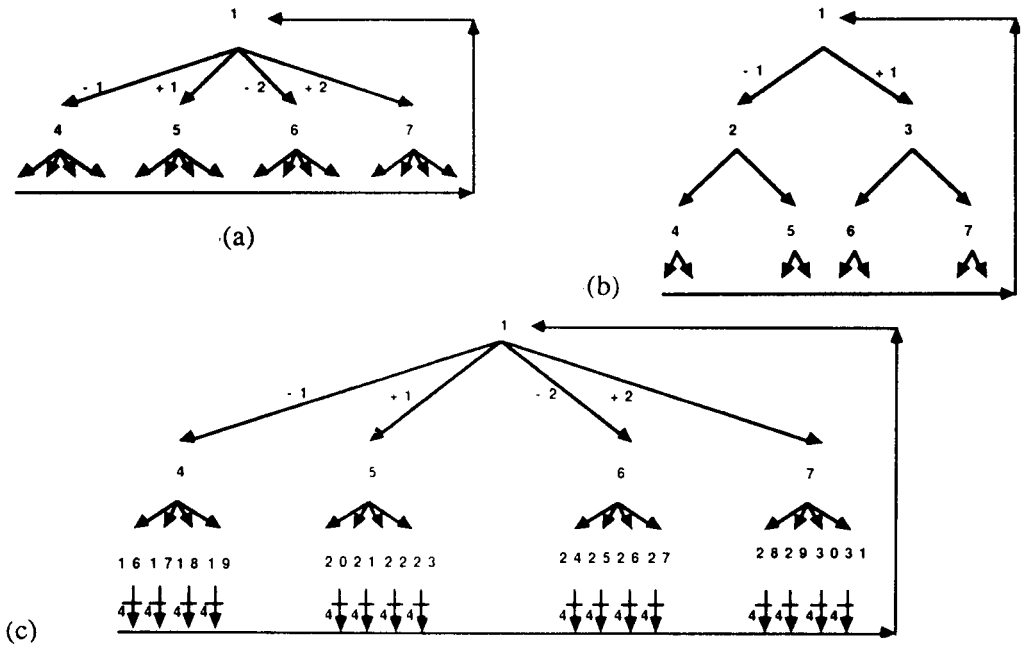

Figure 6. Sample synthetic protocols

Analysis of benchmark structure

Efficient use of the benchmark protocol family requires *a priori* knowledge of the constraints on the final generated global state space. For example, monitoring the boundary conditions of uniprocessor and multiprocessor, main-memory based systems requires a protocol to generate a global state space which is sufficiently large but does not exceed the memory capacity. In parallel verification systems, evaluating communication overhead when large messages are routed, also demands a sufficiently large state space. In characterizing the benchmark family, both the size of the *stable state* space as well as the final global state space are analysed. A single message channel capacity is assumed in the analysis.

As defined in Reference 7, a *steady system* is 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 of the transition in the reception process matches the triggering event in the sending process.

The cardinality of the stable state space $(S_s(d))$ is equal to the initial number of transitions of a single process; namely

$$S_s(d) = \sum_{i=1}^{d+1} m^i = \frac{m - m^{d+2}}{1-m}$$

Let q be the number of all possible *steps* a process pair can initiate from an empty channel, where a step is an event pair such that both process A and B execute a single transition. Therefore, in an m-ary benchmark protocol, $q = 3m/2$. For example, if $m = 4$, then there are two message types, namely 1 and 2, and the possible steps are $\{\langle +1, +1 \rangle, \langle +2, +2 \rangle, \langle +1, -1 \rangle, \langle +2, -2 \rangle, \langle -1, +1 \rangle, \langle -2, +2 \rangle\}$. The cardinality of the global state space ($G_s(d)$) is

$$\begin{cases} (m^2 + 2m), & d = 0 \\ (m^2 + 2m)(q + 1), & d = 1 \\ (m^2 + 2m)(q^2 + q + 1) - \dfrac{(m^4 + m^3 - 2m^2)}{4}, & d = 2 \\ (m^2 + 2m)\displaystyle\sum_{i=0}^{d} q^i - \frac{(m^4 + m^3 - 2m^2)}{4}\left(\sum_{i=0}^{d-2} q^i + \sum_{i=0}^{d-3}\sum_{j=1}^{d-2-i} q^i m^j\right), & d \geq 3 \end{cases}$$

Thus, given any member in the protocol family, the exact number of steady and global states can be computed analytically, and the complexity can be estimated.

Extensions to structure

The protocol family described above forms a basis for analysing verification systems. However, multiple communicating processes, non-symmetrical processes, as well as some protocol properties, e.g. deadlock, unspecified reception and non-executable interaction, are not represented in the benchmark.

Multiple communicating processes and non-symmetrical processes are incorporated into the benchmark in a straightforward manner. By replicating each process *t* -fold, a *t* -process communicating system benchmark is provided. The transition labels must then be encoded to indicate which subset of processors are involved with the transition. Similarly, if a non-symmetric communicating system is desired, some number of subtrees from one or more processes are removed. The pruning of subtrees also incorporates erroneous protocol properties in the benchmark structure, as described below.

To test that the verification system under evaluation detects erroneous *property p,* several structural extensions are proposed. Note that this property behaviour list described below is not exhaustive. The intent is to demonstrate the feasibility of incorporating different protocol errors into the existing structure. The erroneous property definitions are based on those provided in Reference 12.

A *deadlock state* is a global state reachable from the initial global state, with all channels empty, in which no transitions are possible. An *unspecified reception* is a reception that is executable but not specified in the design. Such a reception results in an unpredictable system behaviour. Modifying the benchmark structure so as to incorporate deadlock and unspecified reception states in the resulting global state space is accomplished by removing all *m* links and descendants of node *i.* Note that this results in node *i* having an outdegree of 0.

A *non-executable interaction* is a transmission or reception that is specified in the design but never executed. Non-executable interactions are introduced by modifying all *m* outgoing links of node *i* to point to the root. All other states and transmissions are left unchanged. This results in having all the children nodes of *i,* namely *mi, mi* +l, *mi+2,* . . . . (*i* +l) *m* -1, having an indegree of 0. Thus, states *mi, mi*+ 1, *mi+2,* . . . . (*i*+ 1) *m* – 1 are unreachable from the starting state and hence generate nonexecutable global states.

PERFORMANCE EVALUATION

Invoking the protocol verification system using two slaves and the finite state specification of the *(d*= 1, *m* = 12) synthetic protocol results in the output in Figure 7. The output consists of the number of tuples (transitions) in the steady (156), synchronous global (312), and asynchronous global (3192) transition relations, followed by a listing of the protocol errors detected as part of the verification. In this case, no serious error was detected. However, several of the global states were ambiguous (non-distinguishable).

In addition to the information related to the protocol itself, system oriented information is provided. Namely, both the actual and simulated time for each slave and for the master process are recorded. The number of bytes sent and received by each process is also presented.

The timings presented here were obtained by emulating a hypercube on an Encore Multimax™ running the Umax™ 4·2 operating system, a distributed version of Unix™. The Multimax configuration-consisted of 18 processors and 128 megabytes of memory. Each logical hypercube node was actually a Multimax process and

™ Multimax is a trademark of Encore Corp.; Umax is a trademark of Encore Corp.; Unix is a registered trademark of Unix International.

```
**************************
Dimension I output
**************************

**Card(J) is 156
** Card(T) is 312
** Card(F) is 3192
!! No Deadlock States Detected
!! No Nonexecutable Transmission for A
!! No Nonexecutable Transmission for B
! ! No Nonexecutable Reception for A
!! No Nonexecutable Reception for B
! ! Ambiguity State Exists
!! Protocol incomplete


STATS for total run
-----------------------------------win slave 0 timing stats ----------------------------
User time          = 10475.417s      System time          = 18.467s
User Send time     = 103.767s        Sys Send time        = 0.767 s
User Rcv time      = 268.717 s       Sys Rcv time         = 4.083 s
User Read time     = 0.213 s         Sys Read time        = 0.070 s
Send count         = 828             Bytes sent           = 1826252
Rcv count          = 733             Bytes rcvd           = 2099376
Simulated Time     = 7418.265 s
-----------------------------------end slave 0 timing stats ----------------------------


-----------------------------------start slave 1 timing stats ----------------------------
User time .        = 10476 .383s     System time          = 17.583 s
User Send time     = 15.067s         Sys Send time        = 0.267 s
User Rcv time      = 143.867s        Sys Rcv time         =0.717 s
User Read time     = 0.267 s         Sys Read time        = 0.067 s
Send count         =721              Bytes sent           = 1883564
Rcv count          = 1208            Bytes rcvd           = 2055152
Simulated Time     = 7241.583 s
-----------------------------------end slave 1 timing stats ----------------------------


-----------------------------------start master timing stats ----------------------------
User time          = 10480.850s      System time          = 13.933 s
ChidrenUsertime    = 0.000 s         Children System time = 0.000 s
User Send time     = 4.550 s         Sys Send time        = 0.383 s
User Rcv time      = 10475.717 s     Sys Rcv time         = 12.783 s
Send count         = 753             Bytes sent           = 453376
Rcv count          =361              Bytes rcvd           = 8664
Simulated Time     = 2.029 s
-----------------------------------end master timing stats ----------------------------
done global_continue = T
```

Figure 7. Sample protocol verification system prototype output

executed on a dedicated node. As viewed by the verification software, each node comprised its own separate memory and disk storage, and inter-node communication was restricted by the emulation software to support only the nearest-neighbour communication channel of hypercube systems. Neglecting the significant multiplicative overhead involved in emulating a hypercube system, the timings produced are indicative of the expected behaviour in performance obtained by running the verification software on a 16-node hypercube.

(Only the timings obtained using the complete verification software are presented.

That is, no results demonstrating the effects of the various data redistribution primitives (Section 3.1) on the performance of the underlying database operations are given. For the results from such experimentation, the reader is referred to Reference 17.)

Two protocols, namely a modified Bi-Synch [3] and X.21, [15] were verified using the developed system. Timings on a 2, 4, 8 and 16 node emulated hypercube are presented in Figures 8(a) and (b). As shown in Figure 8(a), minimal reduction in the computation (verification) time resulted from parallelism in the verification of the X.21 protocol. The minimal improvement resulting from the small problem size. The final asynchronous global relation F for the X.21 protocol comprises of only 238 transitions. As the problem size increases, as in the modified Bi-Synch case (Figure 8(b)), the reduction in processing time due to parallelism becomes more significant. The Bi-Synch protocol verification results in a final asynchronous global relation *F* comprising 354 transitions.

To evaluate the scalability of the developed system, three synthetic protocols were verified. Protocols with *d–m* parameters of (*d*= 1, *m* = 12), (*d* =2, *m* =6), and (*d*= 3, *m* = 4)were selected according to the size of their respective final asynchronous global relation *F*, 3192, 4008, 5424, respectively. Figure 9 illustrates the times associated with verifying each protocol on 2, 4, 8 and 16 node systems. Note that in the verification of the (*d* =3, *m* =4) protocol, nearly a sixfold reduction in the overall verification time was observed when a factor of 8 times the number of nodes were employed.

The analytical analysis of this system described in Reference 6, postulated the effective use of over 100 processors for protocol verification of highly complex protocols. Unfortunately, as the facilities were limited, this postulate could not be

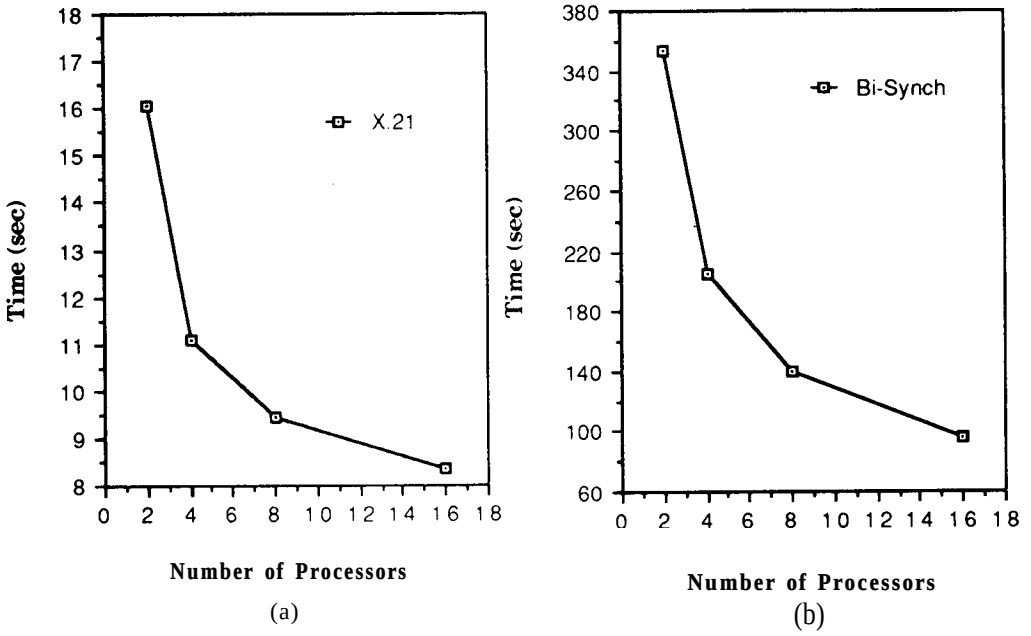

Figure 8. Verification times of existing protocols: (a) X.protocol verification: (b) Bi-Synch protocol verification

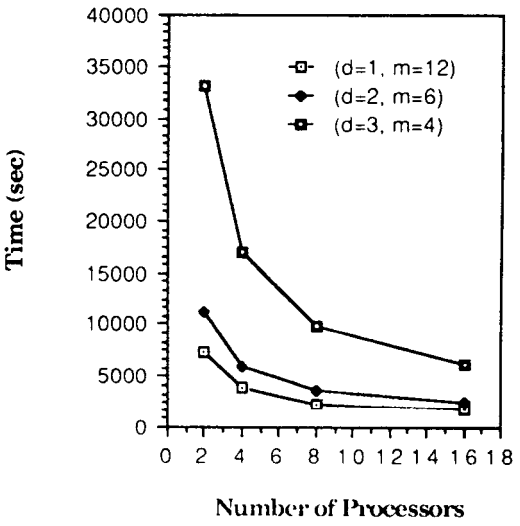

Figure 9. Verification times of synthetic protocols: evaluation via synthetic protocols

verified. However, based on the limited runs we ran, it seems likely that for highly complex protocols encountered in such applications as telecommunication and distributed database systems, effective exploitation of parallelism is possible and is likely to be required to verify the protocols.

CONCLUSION

Protocol verification requires great computational resources. To meet the needed computational demands, we proposed, implemented, and evaluated a prototype of a parallel, database-driven protocol verification system. A description of our system was presented.

To evaluate our system we proposed and analysed a family of protocols as a benchmark for protocol verification systems. Using this benchmark, we evaluated our system and demonstrated the potential of exploiting parallelism in verifying complex protocols. We also used our system to verify two simple conventional protocols.

Our implementation used the Argonne National Laboratories' macro package to abstract out the machine dependencies. Unfortunately, the overhead incurred in emulating the hypercube multicomputer using these macros rendered our measurements meaningless in terms of absolute execution times. Only a system scalability evaluation could be performed using our timings.

As part of future work, we plan to modify our verification system to support the verification of protocols specified in the extended finite state automata model. [10] This will be accomplished using inline procedure invocation within the actual database, similar to the approach described by Stonebraker, *et al.* [23] We will also investigate the parallelization of the actual verification algorithm itself. This parallelization is in addition to what has previously been done in terms of the basic relational operators of which the verification algorithm is composed. Currently, the verification

algorithm is a serial driver with the underlying relational operations being evaluated in parallel. With a parallel driver invoking parallel primitives, we hope to extend the range of complexity of the communication protocols that can be verified using our system. Finally, we plan on implementing this system on an actual hypercube.

ACKNOWLEDGEMENTS

I graciously acknowledge the software development efforts of Nick Karonis and Paul Jackson, the analytical assistance of Tom Richardson, and guidance by Tony Lee.

This research was partially supported by the U.S. National Science Foundation under contract No. CCR-91-09804 and the Virginia Center for Innovative Technology under contract No. INF-91-010. This work used the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by DARPA/RADC contract No. F306002-88-C-0031. (Portions of the described effort were developed while the author was at Bellcore.)

REFERENCES

1. S. Aggarwal, R. Alonso and C. Courcoubetis, 'Distributed reachability analysis for protocol verification environments', in *Discrete Event Systems: Models and Applications,* Lecture Notes in Control and Information Science, Springer-Verlag, 1987, 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 Engineering.,* **SE-14** (3) **,** 280–290 (1988).
3. C. H Chow, M. G. Gouda and S. S. Lam. 'A discipline for constructing multiphase communication protocols', *ACM Trans. Computer Systems,* **3** (4), 315–343 (1985).
4. C. H. Chow and S. S. Lam, 'PROSPEC: an interactive programming environment for designing and verifying communication protocols', *IEEE Trans. Software Engineering,* **SE-14** (3), 327–338 (1988).
5. D. M. Cohen and T. M. Cuinther, 'The IC* system for protocol development', *Proceedings of the ACM SIGCOMM '87,* August 1987.
6. O. Frieder and G. E. Herman, 'Protocol verification using database technology', *IEEE J. Selected Areas in Communications,* **SAC'7** (3), 324–334 (1989).
7. T. T. Lee and M. Y. Lai, 'A relational algebraic approach to protocol verification', *IEEE Trans. Software Engineering,* **SE- 14** (2), 184–193 (1988).
8. D. P. Sidhu and T. P. Blumer, 'Verification of NBS class 4 transport protocol', *IEEE Trans. Communications,* **COM-34** (8), 781–789 (1986).
9. T. Suzuki, S. M. Shatz and T. Murata, 'A protocol modeling and verification approach based on a specification language and Petri nets', *IEEE Trans. Software Engineering,* **SE-16** (5) (1990).
10. T. Y. Choi, 'Formal techniques for the specification, verification, and construction of communication protocols', *IEEE Communications Magazine,* **23** (10), 46–52 (1985).
11. R. Miller, 'Protocol verification: the first ten years, the next ten years: some personal observations', *CESDIS Technical Report,* Goddard Space Flight Center, No. TR-90-14, 1990.
12. M. C. Yuang, 'Survey of protocol verification techniques based on finite state machine models', *Proceedings of Computer Networking Symposium, 1988.*
13. G. Holzmann, *Design and Validation of Computer Protocols,* Prentice Hall, Englewood Cliffs, New Jersey, 1991.
14. C. K. Baru and O. Frieder, 'Database operations in a cube-connected multicomputer system', *IEEE Trans. Completers,* **C-38** (6) 920–927 (1989).
15. C. W. West and P. Zafiropulo, 'Automated validation of a communications protocol: the CCITT X.21 recommendations', *IBM J. Research and Development,* **22** (1), 60–71 (1978).
16. E. Lusk, R. Overberk. J. Pattersen, R. Stevens, J. Boyle, R. Butler, T. Disz. B. Glickfeld. *Portable Programs for Parallel Processors,* Holt, Reinehart and Winston, Inc. 1987.
17. O. Frieder, 'Multiprocessor algorithms for relational database operators on hypercube systems', *IEEE Computer,* **23** (11), 13–28 (1990).
18. D. Maier. *The Theory of Relational Databases,* Computer Science Press. Rockville. Maryland. 1983.

19. N. Karonis, 'Using the relational algebra for complete protocol verification', *CIS Technical Report 89-10,* Syracuse University, 1989.
20. J. C. Gibson. 'The Gibson mix', *IBM Technical Report TR00.2043,* June 1970.
21. P. B. Hawthorn and D. J. DeWitt. 'Performance analysis of alternative database machine architectures'. *IEEE Trans. Software Engineering,* **SE-8** (l), (1982).
22. M. F. Morris and P. F. Roth, *Computer Performance Evaluation,* Van Nostrand Reinhold, New York, 1982.
23. M. Stonebraker, J. Anton and E. Hanson, 'Extending a database system with procedures', *ACM Trans. on Database Systems,* **12** (3), 350–376 (1987).