DISTRIBUTED REACHABILITY ANALYSIS FOR PROTOCOL VERIFICATION ENVIRONMENTS

Sudhir Aggarwal
Rafael Alonso
Costas Courcoubetis

CS-TR-110-87

August 1987

Distributed Reachability Analysis for Protocol Verification Environments

Sudhir Aggarwal

Bell Communications Research
435 South Street
Morristown, NJ 07960

Rafael Alonso

Department of Computer Science
Princeton University
Princeton, NJ 08544

Costas Courcoubetis

AT&T Bell Laboratories
600 Mountain Av.
Murray Hill, NJ 07974

ABSTRACT

A topic of importance in the area of distributed algorithms is the efficient implementation of formal verification techniques. Many such techniques are based on coupled finite state machine models, and reachability analysis is central to their implementation. SPANNER is an environment developed at AT&T Bell Laboratories, and is based on the selection/resolution model (S/R) of coupled finite state machines. It can be used for the formal specification and verification of computer communication protocols. In SPANNER, protocols are specified as coupled finite state machines, and analyzed by proving properties of the joint behavior of these machines. In this last step, reachability analysis is used in order to generate the "product" machine from its components, and constitutes the most time consuming part of the verification process. In this paper we

investigate aspects of distributing reachability over a local area network of workstations, in order to reduce the time needed to complete the calculation. A key property which we exploit in our proposed design is that the two basic operations performed during reachability, the new state generation, and the state tabulation, can be performed asynchronously, and to some degree independently. Furthermore, each of these operations can be decomposed into concurrent subtasks. We provide a description of the distributed reachability algorithm we are currently in the process of implementing in SPANNER, and an investigation of the scheduling problems we face.

July 31, 1987

Distributed Reachability Analysis for Protocol Verification Environments

Sudhir Aggarwal

Bell Communications Research
435 South Street
Morristown, NJ 07960

Rafael Alonso

Department of Computer Science
Princeton University
Princeton, NJ 08544

Costas Courcoubetis

AT&T Bell Laboratories
600 Mountain Av.
Murray Hill, NJ 07974

1. Introduction

Designing reliable distributed software such as computer communication protocols is extremely difficult and challenging. Informal specifications of such software are often imprecise and incomplete, and are not sufficient to ensure correctness of many simple distributed algorithms. One reason is that the concurrent execution of components typically results in an exploding number of execution histories. This makes the prediction of all possible erroneous behavior of such systems prohibitively complex for the human mind, and as a result the designer *must* rely on formal methods for specifying and analyzing the software. There is an increasingly extensive literature on such formal methods and tools; for example, see [Bo87] for a survey.

Among the formal specification methods, finite state models are one of the most popular. In these models, the system is described as a set of coupled finite state machines (FSMs), each machine modeling a concurrently executing component. The reason FSMs are widely used to describe complex systems is that it is conceptually easier to describe

such a system in terms of a large number of small components, and then derive the complete system by taking the "product" of these components. In addition to this, in many cases, FSM descriptions can be directly translated to implementable code or implemented in hardware, see [AC85, AK84, GK87].

There are many existing tools for the analysis of finite state models, for example see [ABM86, An86, Fl87, CG86]. The way these tools work can be summarized in the following steps. First, they provide an environment in which the designer specifies the FSMs by describing the components of the system. Usually, the designer also specifies the task that must be satisfied by the system in order to ensure correct execution. In the second step, the system uses the description of the components to construct the "product" FSM which models the complete system. This step constitutes the reachability analysis of the system, during which a database of all the reachable states and transitions of the product FSM is constructed from the specifications of the component FSMs. The last step consists of checking the validity of the task formula on the product FSM. This can be accomplished in a "partial" way by assigning probabilities to the FSM and checking correctness on some finite set of most probable histories, obtained by simulation, or in a complete way by doing model checking of the FSM and the task formula, see [CE82, QS82]. In some systems, the last step is embedded in the second step, and corresponds to doing reachability analysis on a larger number of FSMs, some of which model the task requirements, see [ACW86]. One can also use various reduction techniques depending on the underlying model, so that the product FSM is substituted by a smaller one. Further details about these techniques can be found in the literature, and are beyond the scope of this paper.

From the previous discussion, it follows that a limiting factor for the practical application of the FSM methods is the time it takes to perform the reachability analysis. With the current technology, graphs of 10^4 to 10^6 states can be analyzed in times on the order

of minutes or hours, by running the tools on single dedicated workstations. In this paper we focus on how to move this limit substantially further by performing the reachability analysis in parallel. Note that an important issue in favor of distributing the reachability analysis is the size of the table of the explored states. For large graphs, this table cannot fit in the main memory of a single workstation and the tabulation becomes increasingly slower as the number of explored states grows. We describe a parallel reachability algorithm and its implementation aspects, for the already existing tool SPANNER [ABM87], in a local area network of SUN workstations. We should emphasize that the design we propose is general enough to be used for parallelizing the reachability analysis in most of the existing FSM tools.

We will elaborate now further on the ideas presented in the paper. A "centralized" reachability analysis is performed as follows. The input consists of the description of N FSMs. A "global" state is a state of the product FSM, and consists of a vector of N local states, one per component machine. The underlying model provides a way to compute for each global state, the set of all possible successor global states. The reachability analysis starts with some initial global state, and completes when all global states reachable (in any number of steps) from this initial state, are found. While doing this, there are two basic operations involved: the *state generation*, which given a global state, computes its successor global states, and the *state tabulation*, which given a global state, checks if it has been already found by keeping an updated table of the global states visited so far. The distributed reachability analysis we propose is basically performed as follows. There are *n state generators* and *m state tabulators*. Each state generator receives (global) states from the tabulators, computes their successor states, and sends them to the tabulators. A state whose successor states have been computed is considered to be "explored". The tabulators receives newly generated states from the generators, filter out the states that have already

be found, and send unexplored new states to the generators. The key issues we address in our design is how to distribute the newly found state information among the tabulators, and the scheduling of the work requests among the tabulators and the generators, so that the workload of different processors remains balanced. As it turns out, the scheduling problem involved is non trivial, and has many generic aspects. This is due to the large number of messages, and the comparable magnitude of the time involved in processing the work carried by a message, with the message delay of the network.

The paper is organized as follows. In section 2, we briefly describe the SPANNER system and the model of FSMs on which SPANNER is based. In section 3 we describe the design of the distributed software, and its implementation environment. In section 4 we examine the underlying scheduling problem, and we provide a queuing model for the system. This model can be used as a basis for simulating the performance of the system, with different scheduling parameters. We also mention two open scheduling problems which abstract different parts of the original problem and seem interesting for further research. At the end of section 4, we investigate performance issues related with the gain in speed of the reachability analysis due to parallelism. In section 5 are the conclusions of this work.

2. The Selection/Resolution model and the SPANNER system

2.1. The Selection/Resolution model

For completeness, we review the selection/resolution model and the SPANNER system. For simplicity we discuss the model in terms of its operational semantics. Further details are available in [GK82, AKS83, ABM87]. The selection/resolution model is a formal method of describing a complex system as a finite set of coupled FSMs. Each component FSM (called a process) is specified as an edge labelled directed graph, see figure

2.1.

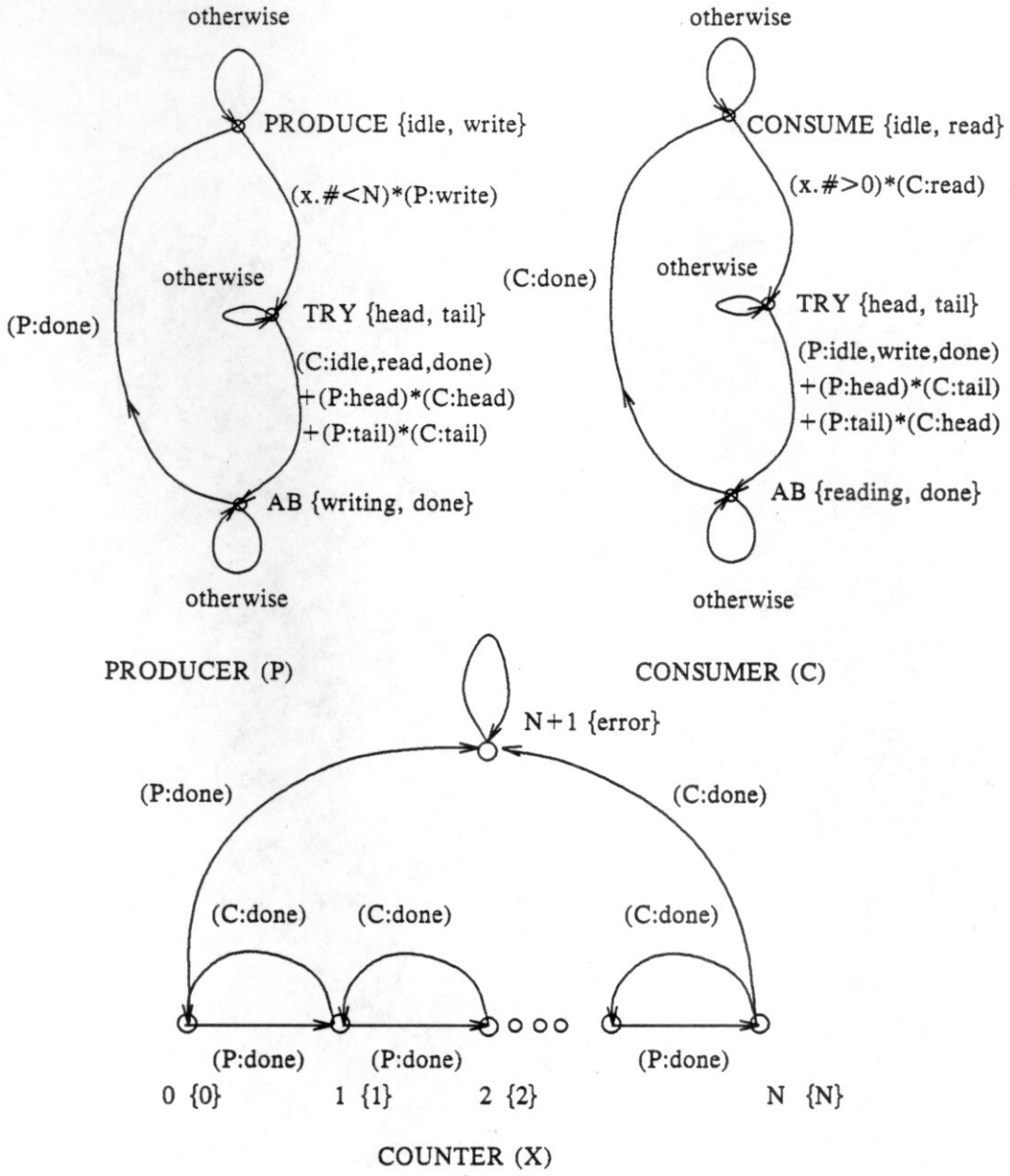

Figure 2.1

The vertices of the graph are *states* of the process, and the directed edges describe a state transition that is possible in one time step. A state encapsulates the past history of the process and is private to that process. That is, no component FSM can know about this state directly. In each state, a process can nondeterministically choose from a set of *selections* (enclosed in braces next to the state). The selections are signals processes use to

coordinate. They can be viewed as indicating the "intention" of the process. The component FSMs use these selections to determine their transitions.

The directed edges of the component FSMs are labelled by elements of a Boolean algebra generated by the selections of the processes. We use * to indicate the multiplication operator (Boolean *and*), and we use + to indicate the addition operator (Boolean *or*). We use ¯ for the Boolean *negation*.

After each process has made its selection, each process decides on a transition to a new state. This *resolution* is done as follows. First, calculate the global selection of the processes. This is done by multiplying together the current selections of all the processes. Note that, by definition, this product is the *and* of all the current selections. Next, each component FSM independently determines which transitions out of the current state are enabled. It determines if a transition is enabled by the global selection by multiplying the edge label by the global selection and checking if the result is 0 in the Boolean algebra. If the product is 0, the transition is not enabled. Otherwise, it is enabled (a valid transition). Finally, each process chooses one of the enabled transitions and transitions along that edge to its new state.

Consider the case of k processes $P_1, \ldots, P_k$ and let the selection of P_i be s_i at time step t. Then the global selection $s = s_1 s_2 \ldots s_k$ is the *AND* (in the Boolean algebra) of the individual selections. If process P_i is in state v at time step t, and the label on the edge from v to w is ℓ, then w is a possible state at time $t + 1$ if $s \cdot \ell \neq 0$.

A *chain* of a process is a sequence of state-selection pairs consistent with the dynamics described above. Intuitively, a chain is a sample path of the behavior or possible history of the process, where at each time step we record the state and selection of the process.

2.2. The Spanner System

SPANNER is an environment consisting of a set of modules for specifying and analyzing protocols. The underlying formal model is the selection/resolution model discussed above. SPANNER allows the user to specify a protocol as a set of coupled FSMs using the SPANNER specification language. The parser module checks the specification for syntactic correctness, and produces an intermediate description used by other modules.

The basic construct of the specification language is a process; this corresponds to a labelled directed graph of the s/r model. The initial declaration of the process simply describe the ranges of states and selections and gives the user the option (using the keyword **valnm**) of providing descriptive names for the states and selections. The *import* declaration describes which processes' selections are visible in that process. The *init* declaration declares the initial state of that process. The *trans* section is the transitions section and consists of blocks that define transitions from sets of states. The format of a block is shown in figure 2.2.

```
current state      {selection list}
  > next state     : condition;
      .                .
      .                .
  > next state     : condition;
```

Figure 2.2

Figure 2.3 shows the processes of a simple producer-consumer problem in the specification language.

SPANNER provides a variety of other constructs that make it easier to specify large

```
constants N = 3

process P        /*the producer*/
 import C
 states 0..2     valnm [PRODUCE:0, AB:1, TRY:2]
 selections 0..5 valnm [idle:0,write:1,writing:2, done:3, head:4,tail:5]
 init PRODUCE
 trans
  PRODUCE        {idle, write}
    > TRY        :write);
    > $          :otherwise;

  TRY            {head,tail}
    > AB         :(C:idle,read,done) + (P:head)*(C:head) + (P:tail)*(C:tail);
    > $          :otherwise;

  AB             {writing, done}
    > PRODUCE:(P:done);
    > $          :otherwise;
end

process C        /*the consumer*/
 import P
 states 0..2     valnm [CONSUME:0, AB:1, TRY:2]
 selections 0..5 valnm [idle:0, read:1, reading:2, done:3, head:4, tail:5]
 init CONSUME
 trans
  CONSUME        {idle, read}
    > TRY        :(X.# > 0) * (C:read);
    > $          :otherwise;

  TRY            {head,tail}
    > AB         :(P:idle,write,done) + (P:head)*(C:tail) + (P:tail)*(C:head);
    > $          :otherwise;

  AB             {reading, done}
    > CONSUME:(C:done);
    > $          :otherwise;
end

process X        /*the counter*/
 states 0..N+1   valnm [ERROR:N+1]
 selections 0..N+1 valnm [error:N+1]
 init 0
 trans
   $             {$}
    >($ + 1)%(N+2): (P:done);
    >($ - 1)%(N+2): (C:done);
    > $          : otherwise;
end
```

Figure 2.3

systems. This includes the notion of *cluster*, to facilitate hierarchical development, and the notion of *process type* as a template for the instantiation of similar processes. These constructs are discussed in the references.

The SPANNER system allows the user to experiment with and study the system of coupled FSMs in two ways. First, the system can be studied using the reachability graph. The reachability graph is a graph whose vertices are global states (vectors of local states), and whose directed edges are valid transitions between global states.

The latest version of SPANNER is actually based on an extension of the s/r model which allows reasoning about infinite paths (chains) [ACW86]. It turns out that many questions about protocols such as deadlocks, livelocks, and liveness can be answered solely by proper investigation of the reachability graph. The general mechanism that we use is to add monitor processes that either check for or ensure certain properties of interest. For example, to an existing protocol, we could add a process that ensures that a particular component always makes progress and does not pause forever (a liveness property). Similarly, we could add a process that checks that a particular task such as receiving messages in the order sent was met. In this approach, proving the validity of arbitrary temporal logic formulas is done by checking properties of a reachability graph.

In order to make it convenient to study the reachability graph, SPANNER produces a database that consists of three tables (relations). The *global reachable states table* (table *r*) has as attributes *index* (the number of the global state), and the *local state* for each process identified by the name of the process. In addition, each global state has the attribute *cc* that identifies to which strongly connected component that state belongs. The transitions table (table *t*) has as attributes *to-state* and *from-state* that specify the global state numbers for the one step transitions. Using a set of commands, the user can query the relations to determine those table entries that satisfy a particular condition. For example, in the

producer-consumer protocol of figure 2.1, we could ask if there is a global reachable state with process *P* in state *AB* and process *C* in state *AB*, corresponding to both processes accessing a common buffer at the same time, and we would find that the answer is no. In addition, the database has a third relation called table *c* that is used in checking for liveness properties. For details, see [ABM88].

Another way of studying the system of coupled FSMs is through simulation. This is particularly useful for very complex protocols, since interesting constraints can be imposed on the simulation. For example, it is possible to assign probabilities to the selection choices and it is also possible to force a selection to be held for a particular number of time steps. SPANNER allows the creation of a database of sample runs using a set of simulation modules. These modules allow a simulation to be setup, using the constraints mentioned, then simulated, and finally analyzed. The user can analyze the results of the simulation by querying the generated database using an interactive query language, similar to querying the reachability database.

Reachability

Reachability in SPANNER is done in a fairly standard way. Given, a global state of the system (a vector of local states of components), the first step is to generate potential new global states. The is done by checking for all possible transitions that are enabled in each component for that given global state, and then looking at the new local states that result. These new local states are combined to form the next set of global states. This is the *generator* function of reachability. Next, the potential new states are checked to see if they truly are new states. This is done by keeping the reached states in a hash table. This is the *tabulator* function of reachability. Unexplored states are kept on a list of states to be investigated, and they can be explored by either breadth-first or depth-first search.

In this paper, we essentially discuss various ramifications of parallelizing this

reachability algorithm. As noted in the introduction, states can be generated in parallel, since the generation of next states from two different initial states can be done independently. Thus, the generator function can be done in parallel. Further, by being careful to handle only parts of the hash table, the tabulator function could also be done in parallel. It should also be noted that the generation of new states from a given initial state can also be made parallel to some extent, since each component process can independently determine the enabled transitions from its initial local state.

3. Implementation

In this section we provide some of the details of our proposed implementation. First, we describe the particular environment in which we intend to construct the parallel version of SPANNER. We then outline the distributed reachability algorithm. The details of the scheduling policy implemented are discussed in section 4.

3.1. Implementation Environment

The system will be implemented on a network of SUN workstations in the Distributed Computing Laboratory of Princeton University. We will use SUN models 2 and 3, which will provide us with some heterogeneity with respect to processor speeds (the model 3 processor is significantly faster than that of the model 2). The machines are connected via a dedicated Ethernet network, which ensures that during our experiments the network is not loaded by extraneous messages.

The machines will be running SUN UNIX version 3.3, which supports a variety of networking protocols [Sun86]. The currently implemented protocols are either *stream* or *datagram* oriented. Stream protocols provide a bidirectional, reliable, sequenced communication channel between processes. The stream protocol implemented by SUN is the TCP protocol defined by DARPA [Po80a]. Datagrams also allow bidirectional communication,

but make no delivery guarantees. Datagram packets may be lost, duplicated, etc. SUN's datagram implementation is based on the IP protocol standard [Po80b]. Either type of protocol can support a message rate of somewhat less than 1 megabit per second between two SUN workstations on an otherwise idle Ethernet.

At first glance it would seem that choosing a stream protocol might be the obvious course of action for our work. After all, users of stream protocols do not have to concern themselves with the details of packet formation, dealing with duplicates, ensuring that messages are not lost, etc. However, this functionality comes at the cost of lessened control over the transmission of data. The user view of a communication stream is that of a boundary-less flow of data. That is, users think of streams as if they were inter-processor versions of UNIX pipes [RT78], and are not aware of the details of the underlying communication layers. Typically however, a stream protocol is implemented on top of a datagram protocol (as is the case for our network). System designers who are primarily concerned with efficiency and performance may need to have access to these underlying layers. For example, it may be desirable to control the amount of data in a datagram packet and the time of its transmission.

In practice, communication systems (i.e., the combination of network hardware, protocols and operating system support) have certain packet sizes that they deliver most efficiently; for example, if the networking code in the operating system kernel needs to move the user's data out of the user's address space before shipping it across the network, a packet size equal to the operating system's page size will usually result in the largest throughput. Another fact that must be taken into account in the implementation of a parallel application is that it is usually more efficient to send one large message than many small ones because there is normally an overhead per packet sent.

In light of the above comments, it seems clear that our choice of networking protocol

is not an obvious one. Our present approach is to develop the initial code using a stream protocol. Once the debugging stage is complete, we will start using the datagram facility, in order to obtain the maximum performance from our system.

3.2. Software architecture

We have already provided some details of the reachability analysis carried out by SPANNER. The distributed version consists of n generators and m tabulators. Each generator stores the complete description of all component FSMs but keeps no information about the set of reachable states. The "global" hashtable (which would be used by SPANNER in its non-distributed version) is now split into m equal nonoverlapping hashtables to be used by each of the m tabulators. Each tabulator has no information about the FSMs, and will only store the global states which hash in its local hashtable. This implies that each global state can be stored in only one among the m tabulators. One can easily define the function h which maps any global state v to the appropriate tabulator $h(v)$ as follows. Compute the hashvalue of v, and check to which of the local hashtables it corresponds. The index of the corresponding tabulator is the value $h(v)$.

A generator is described in terms of three concurrent processes: a receiver process, which feeds the input queue with unexplored states by unpacking the arriving messages; a next state generator, which given a state produces its successor states; and a sender process, which controls the sending of the resulting states through the network. A tabulator is similarly defined in terms of a receiver, a state tabulator, and a sender process. A more precise description follows:

Generator i, i=1,...,n.

Process Receiver

do until *(end of reachability analysis)* {

```
        receive message from network;
        break it into states;
        append the states to the generator input queue
        }
Process Next_State_Generator
do until (end of reachability analysis) {
        get next state v from the generator input queue;
        compute the set next(v);
        for each v' in next(v) do{
                compute j=h(v');
                append <v',v,i> to the generator
                output queue with destination Tabulator j
                }
}
Process Sender
do until (end of reachability analysis) {
        for each output queue with destination Tabulator j, j = 1,...,m, do{
                use the heuristic scheduling policy of section 4
                to pack states into messages sent to Tabulator j

        }
}
```

Tabulator j, j=1,...m.

```
Process Receiver
do until (end of reachability analysis) {
        receive message from network;
```

break it into states;

append these states to the tabulator input queue

}

Process State_Tabulator

variables: U_i is the list of the unacknowledged states

sent to generator i (in the order sent), $i = 1,...n$.

do until *(end of reachability analysis)* {

get next element $<v',v,i>$ from the tabulator input queue;

if v is in U_k for some $k=1,...n$, then

update $U_k =: U_k - (v, v_1, ..., v_s)$,

where $v_1,...,v_s$ are all states in U_k

prior to v;

insert v' in the hash table;

if v' is a new state, append it to the tabulator output queue

}

Process Sender

do until *(end of reachability analysis)* {

Use the heuristic scheduling policy of section 4

to pack states from the tabulator output queue into messages

sent to the generators

}

What we have not specified yet is how the *end of reachability* condition will be detected by the processes. A simple way to do this is the following. When any processor remains idle for more than time t, it triggers a round where all processors respond about their work status. If all of them have empty input queues, then the above condition is

satisfied.

A final point to be made is that at the end of the reachability analysis phase each tabulator will have only a portion of the reachability graph and a final coalescing step will be required in which all the sub-graphs are merged. This coalescing step could also be done in parallel with the tabulation.

4. Scheduling aspects

In this section we examine the scheduling problems which must be addressed by the distributed software described in the previous section. There is a plethora of parameters which are important for the efficient execution of the system. The measure of performance we consider is the total finishing time, i.e., the time at which all the reachable states have been found. Intuitively, this can be minimized if we can keep the work balanced among the various computing resources in the system. Achieving such a balance among the tabulators and the generators constitutes a nontrivial scheduling problem which needs some novel heuristic solution.

We start by examining a simpler system consisting of a single tabulator and n generators. The basic controller of the load of the processors in our system is the tabulator. It is the tabulator's responsibility, once an unexplored (new) state has been found, to ship it to the most appropriate generator among the n available. The following reasons make such a decision very complex. In most LAN's, a message has essentially the same cost (delay) if it contains up to some maximum number of bytes. This implies that sending a message containing one state or some system dependent m_{max} number of states, can be achieved for the same cost. This motivates the batching of a large number of states in the same message. In order to make this possible, states ready to be shipped must be kept in a queue until enough of them accumulate to be batched in a message. The negative side-effect of such a decision is that this can produce idle time for the processors waiting to process

these states. On the other hand, if a small number of states per message is sent, this will result in flooding the network with messages and will increase their delay. The reader should be reminded of the size of the problem being on the order of 10^5 to 10^6 states, which makes batching unavoidable. The optimal size of the batch is a crucial parameter to be determined. Note that batching needs to be done from the generator's side as well.

Another consideration is the following. At the beginning of the computation, the number of unexplored states for most problems will grow exponentially fast, and towards the end it will rapidly decrease to zero. If the tabulator ships exceedingly large amounts of work to the generators towards the last phase of the computation, it is likely that during this last phase the workload of the generators will be unbalanced, since there are not enough new unexplored states generated which the tabulator can appropriately distribute to even things out in the generators. A sensible policy should, in the initial phase of the computation, send large amounts of work to the generators to reduce the probability of them staying idle. Towards the end the policy should keep a store of unfinished work in the tabulator's output buffer, from which increasingly smaller batches of work are to be send to the generators in an attempt to keep their outstanding workload balanced, and to reduce the workload uniformly to zero. Such a policy will minimize the total completion time. A key factor in our system that makes such a policy difficult to implement is the random delay with which observations concerning the workload of the generators are made. The tabulator can only estimate the outstanding work in the generator sites from the information in the messages it receives (the newly generated states arrive together with the identity of their generator site). An important characteristic of our system is that the delay of a message through the network is of comparable magnitude to its processing time (time that the destination processor takes to complete the work associated with the states stored in the message). Finding optimal policies with delayed information is in general

outside the scope of any realistic analysis; hence, a heuristic solution to the problem is the most for which one can hope.

Following the ideas in the previous discussion, a scheduling policy should define the following decisions for the tabulator: *when* to send a message to a generator, *to which generator* to send it, and *how much* work the message should contain. The available information for such a decision is an estimate of the amount of outstanding work of each generator, its processing rate (estimated), the amount of work stored in the tabulator's queue, and the average delay of messages in the network.

The Queueing Model.

For modeling the system we make the following assumptions. First, the graph constructed by the reachability analysis is characterized by a distribution function f_G of the outdegree of its states, and by some upper bound N_G of its number of states. In our model we assume that each state has d next states, where d is distributed with f_G and is independent for different states. We also assume that each newly generated state at time t has probability $r(t)$ of being already visited. There are many ways to describe $r(t)$ as a function of $x(t)$ and N_G, where $x(t)$ is the number of states found up to time t. Different such functions correspond to different types of graphs. One such choice is $r(t) = x(t)/N_G$, which corresponds to graphs with small diameter.

The model describes n generators and a tabulator connected through a LAN, see Figure 4.1.

There are two types of customers in the system: the "simple" customers (single states) and the "batched" customers (many states batched into one message). Simple customers belong to different classes. Each class describes the origin-destination of the customer (for example, $T-G_i$) and whether or not the customer serves as an acknowledgement (discussed below). The class of a batched customer is the description of the class of each simple

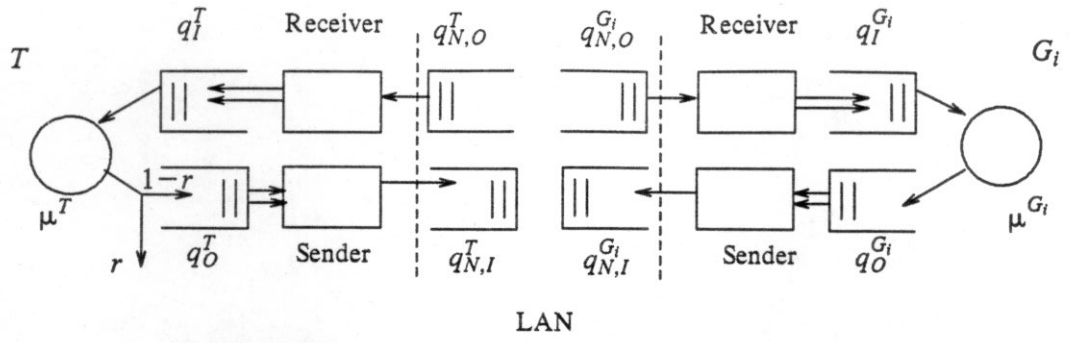

Figure 4.1

customer it contains. Each generator G_i, $i = 1, \cdots, n$, has an input queue $q_I^{G_i}$, an output queue $q_O^{G_i}$, and a server with rate μ^{G_i}. There are two network queues interfacing to the generator, the network output queue $q_{N,O}^{G_i}$, and the network input queue $q_{N,I}^{G_i}$. Customers arriving in the network output queue are of the batched type. Upon arrival, they are automatically "unpacked" (a message containing k states is broken into k single state messages), and the resulting simple customers are placed in the generator input queue. The server serves simple customers from the generator input queue in a FCFS basis, and after each service completion it appends a random number d of simple customers with destination the tabulator, in its output queue. The first among them is tagged as an acknowledgement. As mentioned before, d is distributed with distribution function f_G. A sender processes connects the output queue of the generator to the input queue of the network. Its function consists of making batched customers out of simple customers, and append them at appropriate times in the network queue. Its available information consists of the state of the generator queues, and of some local timer.

The tabulator has an input queue q_I^T, and an output queue q_O^T. These queues interface in a similar fashion to the generators case with a network output queue $q_{N,O}^T$ through an receiver process, and with the network input queue $q_{N,I}^T$ through a sender processes respectively. A batched customer arriving in the network output queue is immediately

unpacked, and the resulting simple customers are placed in the tabulator input queue. A server of rate μ^T serves this queue in a FCFS basis. When a customer finishes service at time t, it leaves the network with probability $r(t)$, and with probability $1-r(t)$ it joins the output queue. In this event, the state counter x is incremented by one, to denote that a new state has been found. If a customer finishing service is an acknowledgment and G_i was its origin (class information of the customer), the variable z_i denoting the outstanding work (number of states to be explored) of generator G_i, is decremented by one. The sender process does the packing of simple customers, assigns the destination of the resulting batched customers, and places them at appropriate times in the network input queue. Its available information consists of the values of x, z_1, $\cdots$,z_n, and the state of the tabulator queues.

We are now left with the description of the network. There are $n+1$ input queues and $n+1$ output queues already mentioned before. The model we choose better describes LAN's of the Ethernet type, such as the one in our implementation. Each non-empty input queue is served with rate $\min[\mu_{\max},\ \mu_{total}/\ \#of\ non\ \varnothing\ queues]$. To model congestion we choose $\mu_{total}<(n+1)\mu_{\max}$. In this model, $\mu_{\max}$ corresponds to the maximum service rate allocated to any network queue. This implies that the minimum delay of a message being in the front of a network input queue, is on the average $1/\mu_{\max}$. If the number of transmitting stations increases, i.e., the number of non-empty network input queues grows, the service rate allocated to a queue decreases as the total network service rate μ_{total} is being shared equally among the competing queues.

Heuristic Scheduling Policies.

A scheduling policy is defined in terms of the algorithms used by the $n+1$ sender processes of the system. We propose a scheduling policy of the following form:

Generator i: The sender process has a timer of duration τ. While the generator

output queue has more than B_G customers, it forms batches of B_G simple customers and delivers them to the network. If there are less than B_G customers, it sets the timer. If the timer times out, and there are still less than B_G customers, it batches them into a message and sends them to the network.

This policy reduces the probability that the tabulator stays idle, while there is work for the tabulator at the generators. It also reduces the number of messages needed.

Tabulator: The sender algorithm uses the following heuristic. The size of the batch is an increasing function of the number of customers in the tabulator output queue q_O^T, starting from zero if the queue is empty, and bounded by some B_T. There is a threshold k^* in the number of customers in q_O^T which affects the operation as follows. If there are more customers than k^*, it continuously makes batches and sends them to the generators, keeping the outstanding work indices $w_i = z_i/\mu^{G_i}$ as close as possible, until an "upper watermark" $W_{\max}$ for each w_i is reached. Then it stops sending, until some w_i drops below, in which case it resumes the sending of work. If the number of customers in q_O^T drops below k^*, it stops sending until for some generator $w_i < W_{\min}$, where $W_{\min}$ corresponds to some "low watermark". Then it resumes sending to this particular generator, until $w_i \geq W_{\min}$. The Readers can convince themselves that in order to achieve an even distribution of customers in the queues of the system, $W_{\max}$ and $W_{\min}$ have to be increasing functions of the number of customers in q_O^T. Choosing $W_{\max}$ to grow appropriately with q_O^T ensures that the tabulator queues do not grow faster than the generator queues. Choosing $W_{\min}$ to decrease as q_O^T decreases provides that in the termination phase all queues will decrease uniformly to zero. The appropriate selection of $W_{\max}$ and $W_{\min}$ remains an open research topic. (A simple queueing theory argument indicates that $W_{\min}$ should be proportional to $(q_O^T)^2$.

We can simply define a policy for the general case of m tabulators by having each

tabulator use $|U_i|$, as defined in the previous section, in place of z_i.

Some Open Scheduling Problems.

There are two simple versions of the model for which the form of the optimal scheduling policy may be more tractable. Solving these could give a greater insight for how to operate the complete system. These models are derived by reducing to zero the transmission delay of the network in one of the directions from G to T or from T to G. The first model is described in Figure 4.2.

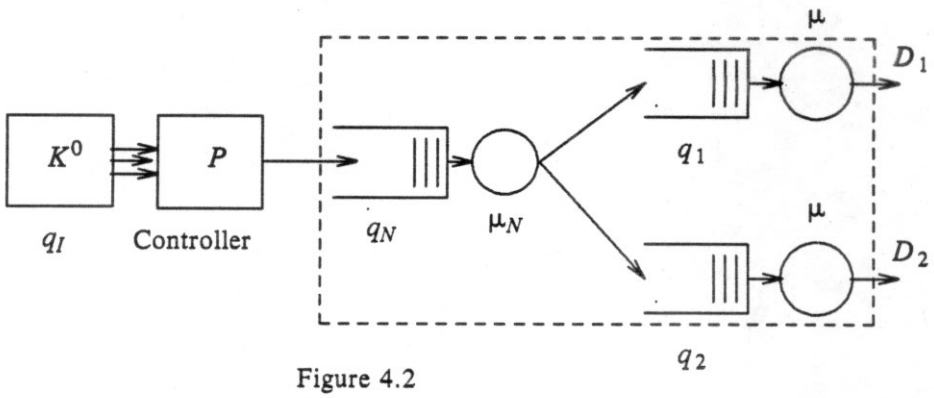

Figure 4.2

There is a finite number K^0 of simple customers at time 0 in queue q_I, There is a sender process (controller) which as in the previous model, makes batches of simple customers, and puts them in the network queue q_N. Each such batch corresponds to a batched customer and is destined to one of the two servers S_1, S_2. The network consists of a server serving with rate μ_N in FCFS basis batched customers. Once a batched customer is served, it joins its destination queue q_i, $i = 1,2$ as the set of its composing simple customers. Each server S_i serves with rate μ from its queue q_i, $i = 1,2$. The information available to the controller is the complete departure process of the system.

The second model is shown in Figure 4.3.

The difference with the previous is that it takes zero delay to append customers in the

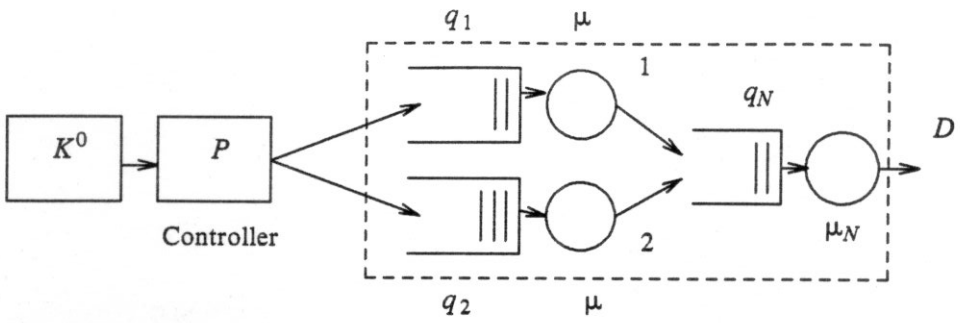

Figure 4.3

queues of the two servers, and that the information available to the controller is a delayed picture of the departure process of the system. For both systems we want to minimize the time all customers complete service. One can easily see that a threshold type policy of the form described in the previous section should perform well in these models. Although the mathematical analysis of these two models might be prohibitively complex, any progress in this direction can result in a valuable practical contribution.

4.1. Performance Analysis

In this section we provide a simple analytic model in order to predict the approximate performance of our scheme. Let

n = number of generator nodes
m = number of tabulator nodes
N = number of states in the graph to be analyzed
d = average out-degree of vertices in state graph
t_g = average time required to generate a single state (seconds)
t_d = average time required to decide if one state is new (seconds)
B = average number of states batched in each message
$t_m(B)$ = average time required to send one message containing B states (seconds)

Using the above definitions, the reachability analysis performed on a single workstation will take total time $T_1 = Nd(t_g+t_d)$ to complete, since each state is examined d times on average, by both the tabulator and generator software. Assuming that using the current technology, a tool runing on a 1 MIP workstation completes a graph of 10^5 states and

$d = 10$ in the order of an hour, we get that $t_g + t_g$ is equal to $5 \cdot 10^{-3}$s.

The first interesting remark is that there is a maximum achievable speed up independent of the number of workstations. To see this, we compute the total time spent in communicating through the network. One can easily see that this time is equal to $N(d+1)t_m(B)/B$. Let B^* correspond to the value of B minimizing $t_m(B)/B$. Then, if an arbitrarily large number of workstation is used, the maximum speed up is aproximately equal to $K_{\max} = B^*(t_d + t_g)/t_m(B^*)$. Using $B = 40$ (25 bytes/state, 1024 byte message), and and network bandwith equal to 5 Mbits/s, we get $K_{\max} = 100$. (For a large number of two-way conversations the effective bandwidth of an Ethernet is at least 5 Mbits).

We examine now how to choose the m, n. Assuming that we have $m + n < K_{\max}$ and we operate in the otimal way (all processors are kept busy until the end), then we must have that $Ndt_d/m = Ndt_g/n = T_1/(n+m)$. From this it follows that the optimal partition is such that $t_d/m = t_g/n$, and the speed up is equal to $m+n$.

5. Conclusions

In this paper we demonstrated that distributed reachability analysis can be easily incorporated into many existing protocol analysis environments and can produce a significant speed up of the analysis. In many research enyironments there is easy access to LANs with 10-20 workstations, which, according to the results of our performance study, is an ideal environment to implement our method. An important remark which makes our approach even more viable in the future is that fiber optic technology makes communication bandwidth available in a faster rate than the rate of increase of hardware speed. In our method a large number of communicating workstations can utilize this available bandwidth.

We are currently implementing the parallel reachability algorithm in SPANNER. We

hope that the results of our implementation will justify the approach presented in this paper.

References

[ABM86] S. Aggarwal, D. Barbara and K. Meth, "Specifying and analyzing protocols with SPANNER", *Proceedings of the IEEE International Conference on Communications*, June 22-25, 1986, Toronto, Canada.

[ABM87] S. Aggarwal, D. Barbara and K. Meth, "SPANNER - A tool for the specification, analysis and evaluation of protocols", to appear in *IEEE Trans. on Soft. Eng.*, 1987.

[ABM88] S. Aggarwal, D. Barbara and K. Meth, "A software environment for the specification and analysis of problems of coordination and concurrency", to appear in the *IEEE Trans. on Software Eng.*, 1988.

[AC85] S. Aggarwal and C. Courcoubetis, "Distributed implementation of a model of communication and computation", *Proc. of the 18th Hawaii Int. Conf. on System Sciences*, January 1985, pp. 206-218.

[ACW86] S. Aggarwal, C. Courcoubetis and P. Wolper, "Adding liveness properties to coupled finite state machines", AT&T Bell Laboratories Technical Memo., 1986.

[AK84] S. Aggarwal and R. Kurshan, "Automated implementation from formal specifications", *Protocol Specification, Testing, and Verification IV*, (Y. Yemini and al. eds.), North Holland, 1984.

[AKS83] S. Aggarwal, R. Kurshan, and K. Sabnani, "A calculus for protocol specification and validation", *Protocol Specification, Testing and Verification III*, H. Rudin and C. West (Eds.), North Holland, 1983.

[An86] J. P. Ansart, et al., "Software tools for Estelle", *Protocol Specification, Testing and Verification VI*, B. Sarikaya and G. Bochman (Eds.), North Holland,1986, pp. 55-62.

[Bo87] G. V. Bochmann, "Usage of protocol development tools: the results of a survey", *Proceedings of the 7th IFIP workshop on Protocol Specification, Testing and Verification*, Zurich, May 5-8, 1987.

[CE82] E. M. Clarke, E. A. Emerson, "Synthesis of synchronization skeletons from branching time temporal logic", *Proc. Logic of Programs Workshop, 1981, Lecture Notes in Comput. Sci.* 131, Springer-Verlag, 1982, 52-71.

[CG86] D. Cohen, B. Gopinath, et al., "*IC**: An environment for specifying complex systems", *Proc. IEEE GLOBECOM Conf.*, Houston, Dec. 1986, pp. 632-637.

[Fl87] A. Fleischmann, "PASS - A Technique for specifying communication protocols", *Proceedings of the 7th IFIP workshop on Prorocol Specification, Testing and Verification*, Zurich, May 5-8, 1987.

[GK82] B. Gopinath and R. Kurshan, "The selection/resolution model for concurrent processes", unpublished.

[GK87] I. Gertner, R. P. Kurshan, "Logical analysis of digital circuits", *Proc. 8th Int'l. Conf. Computer Hardware Description Languages*, 1987, 47-67.

[Po80a] J. Postel, "DOD Standard Transmission Protocol," RFC 761, Information Sciences Institute, January 1980.

[Po80b] J. Postel, "DOD Standard Internet Protocol," RFC 760, Information Sciences Institute, January 1980.

[QS82] J. P. Queille, J. Sifakis, "Specification and verification of concurrent systems in CESAR, *International Symposium in Programming*, LNCS 137, 1982.

[RT78] D. Ritchie and K. Thompson, "UNIX Time-Sharing System," *Bell System Technical Journal*, Vol. 57, Number 6, 1978.

[Su81] C. A. Sunshine, (ed.), *Communication Protocol Modelling*, Artech House, 1981.

[Sun86] "Inter-Process Communication Primer," Sun Microsystems User Documentation, Revision B of February 17, 1986.

[ZWRCB80] P. Zafiropoulo, C. H. West, H. Rudin, D. D. Cowan and D. Brand, "Towards analyzing and synthrsizing protocols", *IEEE Trans. on Comm.*, COM-28, 4 (April 1980), pp. 651-660.