RANDOM WALK TECHNIQUES FOR PROTOCOL VALIDATION

Daniel Barbará
José L. Palacios

CS-TR-221-89

June 1989

RANDOM WALK TECHNIQUES FOR PROTOCOL VALIDATION[†]

Daniel Barbará

C.S. Department
Princeton University
Princeton, N.J. 08544

José L. Palacios

Department of Mathematics [††]
New Jersey Institute of Technology
Newark, N.J. 07102

ABSTRACT

In this paper we present a series of techniques based on random walks to perform state exploration in a reachability graph representing a protocol. Using a set of examples, we show experimental results that demonstrate the usefulness of the techniques. We also present the theoretical framework to prove why some of the techniques work better than others.

June 22, 1989

RANDOM WALK TECHNIQUES FOR PROTOCOL VALIDATION[†]

Daniel Barbará

C.S. Department

Princeton University

Princeton, N.J. 08544

José L. Palacios

Department of Mathematics [††]

New Jersey Institute of Technology

Newark, N.J. 07102

1. INTRODUCTION

The implementation of tools for the validation of computer communication protocols and concurrent processes is a crucial aspect of the design of complex systems today. Researchers have consequently focused a great deal of attention on the development of languages and environments for the formal specification and validation of concurrent processes ([RW83]).

Many of the systems designed for validation of protocols use the idea of *reachabil-*

[†] This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[††] On leave from Universidad Simón Bolivar, Depto. de Matemáticas, Apdo. 89000, Caracas, Venezuela.

ity analysis, i.e., the generation of the graph of all global states reachable from an initial state ([BM80],[ABM88],[SC88],[HK89]). Once this graph is generated, questions can be asked about specific global states and paths. These questions may range from asking whether a particular undesirable global state (e.g., a deadlock) is reachable, to enquiring about a sequence of events in the protocol.

Generating the reachability graph amounts to finding all states that can be reached in one transition from a given state and proceeding in a recursive manner. The states that can be reached in one transition are computed using the protocol specification and the rules of composition for the particular model used.

One problem with this approach is that of *state explosion*, i.e., the size of the reachable graph for real protocols becomes unmanageable. Under these conditions, it becomes impractical to analyze or even to generate the reachability graph.

One way of overcoming these limitations is to generate only a meaningful subset of the reachable states. This subset can be generated for instance by applying random techniques, e.g., generating a random walk over the graph. This idea has been suggested previously in the literature. Piatkowski has suggested the use of random components in the protocol testing strategies [P80]. C. West has reported experiments dealing with the analysis of the Session Layer of OSI by executing a random walk through the reachable state space [W87]. The experiments are aimed to count the number of steps the random walk needs to discover an error in the protocol. No systematic approach of using random exploration has been used or analyzed so far.

A different approach that does not use random state exploration is taken by Maxemchuk and Sabnani in [MS87]. Using a model based in the programming language Com-

municating Sequential Processes (CSP) [H78], they assign probabilities to the transitions in the individual Finite State Machines that compose the protocol. Then, they explore only the most probable section of the protocol according to this assignment. In this way they are able to bound the probability that a particular state not visited in the exploration is reachable. Their idea relies of the belief that if a protocol error (or undesirable state) occurs in a section of the graph that has very low probability, then it might not be worth to modify the protocol by avoiding such error or by adding a recovery procedure.

In this paper we take a first step in proposing and analyzing techniques that use random state exploration, i.e., techniques that proceed to traverse the graph using a random walk. The approach is aimed to search the space for an specific state (or state type), e.g. a deadlock. Along with the techniques, we present the theoretical basis to support their effectiveness. The use of these methods is illustrated by a series of examples. We also use the theoretical framework to suggest new alternative schemas for state exploration.

The layout of the paper is as follows. In Section 2 we discuss the various techniques used for random state exploration. In Section 3, we present the basic notions of the S/R model, used to specify the examples with which we tested the ideas. The random techniques are not dependent of the specific model used for specifying the protocols. The choice of this particular model was based on the availability of SPANNER ([ABM87],[ABM88]), a software environment based on the S/R model, which we modified for our purposes. In section 4 we present the experimental results derived by using the techniques over the examples. In Section 5 we present the theoretical framework for using random walk techniques for state exploration. Finally, in Section 6 we offer some lines of future research and conclusions.

2. RANDOM STATE EXPLORATION

In this section we describe the techniques that we have utilized for random state exploration. These techniques are independent of the model used to specify the protocol. We assume that a particular protocol is made of a set of individual machines, each one represented by a directed graph with labeled transitions. The machines are coupled together by some set of composition rules. Using the rules, one could obtain the reachability graph of global states that defines the protocol. We require that given a global state, we can use the specification to compute the set of next global states to which the protocol can move, without having to generate the whole reachability graph. (Note that this set may include the current global state, i.e., the reachability graph may have a self-loop in this particular state.)

With this condition we have devised three ways of doing random state exploration. In all the techniques, a random walk is performed by selecting randomly at each stage one element from the set of reachable states and making the transition to this state. All of them start with the protocol in a predefined initial state. These techniques can be used to search for a particular predefined state, for instance, a deadlock whose presence in the reachability graph is suspected. In this case the walk stops when the state is reached or when a predefined number of jumps have been performed. Also, the first two techniques can be used to perform a random walk without a particular goal, and to observe if an anomalous behavior occurs. (For instance, the protocol may get stuck in a particular state after a number of moves, indicating the presence of a deadlock.) In this case, the stopping rule for the random walk is the execution of a preselected number of jumps.

In what follows, we describe the techniques.

1) Unrestricted random walk (URW): In each stage of the process with the protocol being in some current global state, generate the set of next reachable states and randomly select one of them with equal probability. Make the transition to this state to continue the walk.

2) Restricted random walk (RRW): Do the same as in 1), but in each stage rule out the possibility of going back to the current global state.

3) Metric-aided random walk (MRW): In this technique, a metric is defined in such a way that we can attach to every global state a positive integer that indicates the "distance" of this state to the particular one we are trying to find. At each state, we sort the set of reachable global states according to the metric and select one of them randomly. The probability of choosing a particular state is inversely proportional to its metric. That is, we tend to select states that are "closer" to the one we are looking for.

The metric used in 3) is defined as follows. Let the system be a set of k machines $M_1, M_2, \ldots, M_k$. Let the goal state be a k tuple $G = <g_1, g_2, \ldots, g_k>$, where each g_i is the local state of the machine M_i. The metric is computed for each particular state $x_i = <s_{i_1}, s_{i_2}, \ldots, s_{i_k}>$, where each s_{i_j} is a local state in machine M_j, as

$$D_i = \max_{j=1}^{k} (d(s_{i_j}, g_j)) \tag{2.1}$$

where $d(s_{i_j}, g_j)$ is the minimum distance between s_{i_j} and g_j in the graph of machine M_j. Let the set of reachable states from x_i be $R_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$. Let D_{ij} be the distance to the goal state from state x_{i_j}, computed with the metric explained above. We can assign to each x_{i_j} a probability p_{ij} of being selected as next state, computed

as:

$$p_{ij} = \frac{\frac{1}{D_{ij}}}{\sum_{k=1}^{m} \frac{1}{D_{ik}}} \tag{2.2}$$

That is, we are favoring in the random selection those states that are "closer" to the goal state, according to the metric defined. Notice that this metric is very inexpensive to evaluate, as opossed to a metric that actually computes the distance to the goal state in the global reachability graph.

Here again at each stage of the walk we can allow the protocol to remain in the current state or impose the restriction of selecting a state different from the current one. Thus, we define the unrestricted metric-aided random walk (MRWU) and the restricted version (MRWR). Equation (2.2) corresponds to the MRWU version, while in MRWR, the probabilities are defined as follows:

$$p_{ij} = \begin{cases} \dfrac{\frac{1}{D_{ij}}}{\sum_{k=1}^{m} \frac{1}{D_{ik}}} & \text{if } i \neq i_j \\ 1 & \text{if } i = i_j \text{ and for all } k \neq i \ (x_k) \notin R_i \\ 0 & \textit{otherwise} \end{cases} \tag{2.3}$$

In the next section we will show some examples on which the three techniques were employed. As we shall see in them, RRW outperforms URW and MRWR outperforms MRWU. In Section 5, we will state the theoretical framework that explains this behavior.

3. THE MODEL AND EXAMPLES

In this section we present the S/R model, used to specify the set of examples with which the methods were tested. We also present the specification of the examples.

3.1. The Selection/Resolution Model

This section reviews the Selection/Resolution model as described in [AKS83]. The Selection/Resolution model provides a mathematical precise way of describing coordination among a set of concurrent abstract modules called *processes*.

Each module in the system is described as an edge labeled directed graph. 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 may be viewed as the encapsulation of past history of the process and is private to it. In each state, a process can nondeterministically choose from a set of *selections*. The selections are essentially signals available to all other processes; they describe what the process ''intents'' to do in the next move.

After each process has made its selection, the ''resolution'' step is taken. That is, the global set of selections is ''multiplied'' with each edge out of the current state thus determining whether the edge is enabled or not. This is possible because edges are labeled by elements of the same Boolean algebra to which the selections belong.

More formally, let L be a Boolean algebra. An *L–process* is a 5-tuple

$$P = (V, S, \sigma, M, I)$$

where

V is the set of states of P

S is the set of selections of P

σ is the *selector* function, $\sigma : V \rightarrow 2^S$

M is the transition matrix, $M : V \; X \; V \rightarrow L$

I is the initial state of P, $I \in V$

The selector function associates with each state s the set of possible selections $\sigma(s)$ that can be made from that state. The transition matrix can be viewed as an adjacency matrix of a directed graph with vertices V where the nonzero entries are labels describing the conditions for a transition to be enabled. Given an edge label $l = M(v,w)$ from state v to state w, if the selection of the process in state v is a, then $a.l \neq 0$ means that the transition to w is possible.

3.2. SPANNER

SPANNER is an environment consisting of a set of software modules (see Figure 3.1) for specifying and analyzing protocols. A user formally specifies a protocol using the specification language. The parser module checks for syntactic correctness and produces an intermediate description used by other modules. The user can generate the reachability graph by using the reachability module. The module produces a database consisting of all global states and transitions. The analysis module allows the user to analyze the graph by querying the database.

We have modified the reachability module of SPANNER to generate random walks instead of the whole reachability graph. For this, in each stage, the module generates all the states that can be reached from the current one, and then selects one of them randomly. We have implemented the three techniques described in Section 2, allowing this selection to be unrestricted (URW), not allowing the return to the current state (RRW) and altering the probabilities with the metric (MRW).

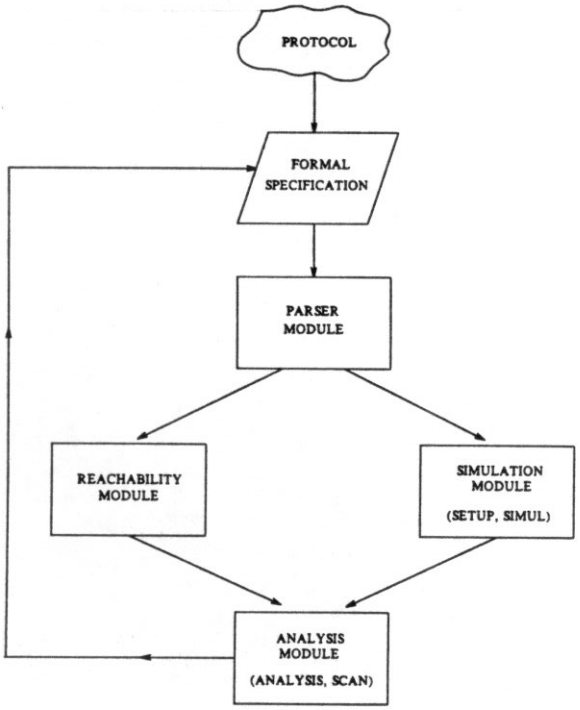

Figure 3.1. The SPANNER Environment

3.3. Examples

3.3.1. The Dining Philosophers.

Our first example is the classical problem proposed by E.W. Dijkstra. Our version of the problem is taken from [H85]. The modeling of this problem with the S/R model is also illustrated in [K85]. Consider a college in which there are *n* eminent philosophers. They spend most of their time thinking, but when they want to eat, they eat in a common dining room. The dining room has a circular table with *n* chairs, one for each philosopher. There are also *n* forks, one to the left of each chair. In the center of the table there is a bowl of spaghetti. In order to eat, a philosopher must use both forks, at her/his left of right. Thus, the behavior of a philosopher is to think, sit down, pick up the left or right fork, pick up the complementary fork, eat, put down the forks, and then stand up and think again. A problem arises if a fork is not available when required by a philosopher. In this case, the philosopher simply waits until it becomes available.

Our model consists of two process types called AFORK and APHIL. The process type AFORK has three states: *ontable, inuseleft,* and *inuseright*. The behavior of the fork is as viewed from the fork's position, and depends on the philosopher to the left and to the right. Both of these processes are of type APHIL. A philosopher is modeled as in the description above. This version of the dining philosophers contains two deadlocks. If all the philosophers sit down and pick the right (left) fork, the system is deadlocked, since none of them will surrender the fork and all will wait for the other to be free. Figure 3.2 shows the specification of this example.

```
typedef process AFORK ( APHIL lphil; APHIL rphil)

states 0..2 valnm[ ontable:0, inuseleft:1, inuseright:2]

selections = states

init ontable

trans

                    ontable
                            >inuseleft      :(lphil:rup) & ~(rphil:lup);
                            >inuseright     :(rphil:lup) & ~(lphil:rup);
                            >ontable        :otherwise;

                    inuseleft
                            >ontable        :(lphil:rdown);
                            >$              :otherwise;

                    inuseright
                            >ontable        :(rphil:ldown);
                            >$              :otherwise;

end

typedef process APHIL ( AFORK lfork; AFORK rfork; APHIL lphil; APHIL rphil)

states 0..7 valnm [ think:0, sitsdown:1, holdlfork:2, holdrfork:3, eat:4,

  lforkdown:5, rforkdown:6, standsup:7 ]
```

selections 0..7 **valnm** [thinking:0, sit:1, idle:2, lup:3, rup:4, eating:5,

ldown:6, rdown:7]

init think

trans

```
think                          {thinking,sit}
        >sitsdown              :(APHIL:sit);
        >think                 :(APHIL:thinking);

sitsdown                       {idle,lup,rup}
        >holdlfork   :(APHIL:lup) & (lfork: ontable) & ~(lphil:rup);
        >holdrfork   :(APHIL:rup) & (rfork: ontable) & ~(rphil:lup);
        >$                     :otherwise;

holdlfork                      {idle,rup}
        >eat         :(APHIL:rup) & (rfork:ontable) & ~(rphil:lup);
        >$                     :otherwise;

holdrfork                      {idle,lup}
        >eat         :(APHIL:lup) & (lfork:ontable) & ~(lphil:rup);
        >$                     :otherwise;

eat                            {eating,ldown,rdown}
        >lforkdown             :(APHIL:ldown);
        >rforkdown             :(APHIL:rdown);
        >eat                   :(APHIL:eating);

lforkdown                      {rdown}
        >standsup              :(APHIL:rdown);

rforkdown                      {ldown}
        >standsup              :(APHIL:ldown);

standsup                       {thinking}
        >think                 :(APHIL:thinking);
```

```
end
process (i = 0..n; philosopher[i]: APHIL ( fork[i], fork[(i+1)%N],
  philosopher[(N+i-1)%N], philosopher[(i+1)%N]))
process (i = 0..n; fork[i]: AFORK ( philosopher[(N+i-1)%N], philosopher[i]))
```

Figure 3.2. The Dining Philosophers

3.3.2. The Elevators.

In this example, we have n elevators that coordinate their movements through a building with m floors. The protocol is designed in such a way that when all the elevators are in the top floor, there is no other transition possible but to remain in that state (a deadlock). The initial state finds all the elevators in the ground floor. This example is interesting in the sense that it provides a way of testing extreme conditions for our techniques. First, by varying the coordination rules among the elevators, one could make it progress very slowly or very fast towards the deadlock. Secondly, since every conceivable state is reachable, one could make the size of the protocol grow very fast by increasing the number of floors in the building. Figure 3.3 shows the first version of the example, called **elevud(n,m)**. In this version, the rules are such that the system has upward drift. In any middle floor, an elevator can only go down if one of its neighbors selects to go up. In the next version, **elevdd(n,m)**, shown in Figure 3.4, the system has downward drift, since for an elevator to go up, its neighbors should remain in their current state or go down. The example *elevdd (n,m)* has been created to test our techniques with a protocol where the "progress" towards the deadlocked state is slow. In this sense it plays a role of an extreme example for which is very difficult to find the deadlock. As we will see this characteristic will play a role in determining a very fast growth on the absorption time (number of steps needed to find the deadlock state) for this protocol as the size of

the protocol increases with *m*.

```
typedef process ELEVATOR ( ELEVATOR lelev ; ELEVATOR relev )

states 0..N selections 0..2 valnm [ stay: 0 , up : 1 , down : 2 ]

init 0

trans

                0                       {stay,up}
                    >0                  :(ELEVATOR:stay);
                    >1                  :(ELEVATOR:up);

                N                       {stay,down}
                    >N-1    :(ELEVATOR:down)&((lelev:up)|(relev:up));
                    >N                  :otherwise;

                $                       {stay,up,down}
                    >$-1    :(ELEVATOR:down)&((lelev:up)|(relev:up));
                    >$+1                :(ELEVATOR:up);
                    >$                  :otherwise;


end

process (i = 0..n; elevator[i]: ELEVATOR ( elevator[(M+i-1)%M] ,
  elevator[(i+1)%M] ))
```

Figure 3.3. The example elevud(n,m)

```
typedef process ELEVATOR ( ELEVATOR lelev ; ELEVATOR relev )

states 0..N

selections 0..2 valnm [ stay: 0 , up : 1 , down : 2 ]

init 0

trans

            0                       {stay,up}
                >0                  :(ELEVATOR:stay);
                >1                  :(ELEVATOR:up);
```

```
N                    {stay,down}
    >N-1    :(ELEVATOR:down)&((lelev:up)|(relev:up));
    >N                  :otherwise;

$                    {stay,up,down}
    >$+1    :(ELEVATOR:up)&(((lelev:down)|(relev:down))
                    |((lelev:stay)&(relev:stay)));
    >$-1            :(ELEVATOR:down);
    >$                  :otherwise;

end

process (i = 0..n; elevator[i]: ELEVATOR ( elevator[(M+i-1)%M] , elevator[(i+1)%M] ))
```

Figure 3.4. The example elevdd(n,m)

3.3.3. Distributed Locking

This example differs from the rest in that there is no deadlock state in the protocol. The example models a protocol for locking all the sites in a network. The node interested in doing the locking initiates the request by sending messages to all the other nodes in the network. Then, the other sites can acknowledge the request positively if no previous lock over the node is in effect. When the requesting node gets all the acknowledgements, it proceeds to enter a critical region. When the critical section is finished, the node sends release messages. Notice that if two sites initiate their requests at times that are sufficiently close from each other, each might get some of the acknowledgments but not all of them. Such potential deadlock is broken by timing out the acknowledgements. The delay in the links is modeled by a set of channel processes (one per site), that choose at each interval whether to deliver the message or to retain it for another interval. The

number of intervals in which this can happen is limited to be D, a predefined constant. (That is, after D intervals, the channel is forced to deliver the message.) We are interested here in finding a state in which one of the sites has reached its critical section while the others are locked. Figure 3.5 shows the specification of this example, called **dlock(n,N,D)**, where n is the number of sites, N the number of units for timeout, and D the maximum delay in the channels.

```
typedef process APROCESSOR (ATIMER mytimer;  predicate request ;
  predicate allacks; predicate release)

states0..4   valnm [idle:0, active:1, locked:2, cs:3, relx: 4]
selections   0..3   valnm [initreq:0, stay:1, ack:2, rel:3 ]
init idle
trans
        idle                          {stay, initreq}
              > active        : (APROCESSOR:initreq);
              > locked    : (APROCESSOR:stay)& ($request);
              > idle                  : otherwise;

        active                        {stay}
              > cs                    : ($allacks);
              > relx                  : (mytimer:timeout);
              > active                : otherwise;

        locked                        {ack}
              > idle                  : ($release);
              > locked                : otherwise;

         cs                           {stay, rel}
              >cs                : (APROCESSOR:stay);
              >idle              : (APROCESSOR:rel);

        relx                          {rel}
```

```
                    >idle          : (APROCESSOR:rel);

end

typedef process ATIMER (APROCESSOR myproc ; predicate allacks)

states 0..N

selections 0..1 valnm[ go: 0 , timeout : 1]

init 0

trans

                    0                   {go}
                        > 1         :(myproc:initreq);
                        > 0          : otherwise;

                    N                   {timeout}
                        >0       : (ATIMER:timeout);

                    $                   {go}
                        >0           : ($allacks);
                       >$+1          : otherwise;

end

typedef process ACHANNEL (APROCESSOR myproc ; predicate release )

states 0..D valnm[ idle : 0 ,pre: D-1 ,  deliver : D]

selections 0..3 valnm[ no : 0 , deliv : 1, delay : 2 , ackr : 3 ]

init idle

trans

                idle                        {no}
                        > 1                 :(myproc:ack);
                        > idle               : otherwise;

                deliver                     {ackr}
                        > idle              :($release);
```

```
                > deliver                   : otherwise;

        pre                                 {deliv,delay}
                > deliver   :(ACHANNEL:deliv)|(ACHANNEL:delay);

        $                                   {deliv,delay}
                > deliver           :(ACHANNEL:deliv);
                > $+1               :(ACHANNEL:delay);

end

process (i = 0..n; P[i]: APROCESSOR( T[i],
  (|; j = 0..n; (P[j]:initreq) & (j != i)) ,
  (&; j = 0..n; (C[j]:ackr)|(j = i)),
  (|; j = 0..n; (P[j]:rel)&( j!= i))))
process (i = 0..n; T[i]: ATIMER ( P[i],
  (&; j = 0..n; (C[j]:ackr)|(j = i))))
process (i = 0..n; C[i]: ACHANNEL ( P[i],
  (|; j = 0..n ; (P[j]:rel)& (j != i))))
```

Figure 3.5. dlock(n,N,D)

4. THE RESULTS

In this section we present the results of applying the techniques described in Section 2 to the examples of Section 3.

We begin with the results of applying the techniques to the example of the dining philosophers (*dinephil* [4].)

Technique	Absorption time
URW	91.25
RRW	64.75
MRWU	15.70
MRWR	13.95

Table 4.1 *dinephil[4]*

The results reported correspond to the average absorption time that our program took to find the deadlock. The number of runs in each case was 20. Two points are worth noticing. First, there is a notable reduction on the absorption time from URW to RRW and from RRW to to cases when the metric is used. Secondly, in all cases the absorption time is a small fraction of the total number of states in the protocol (1022).

The next table presents the resuts obtained for *elevud* (2,3) and *elevdd* (2.3), in each case the average absorption time reported. The number of runs in each case was again 20.

Technique	*elevud* (2,3)	*elevdd* (2.3)
URW	6.35	16.75
RRW	5.7	16
MRWU	3	6.35
MRWR	2.55	6.25

Table 4.2 *elevud(2,3)* and *elevdd(2.3)*

In this table we also see the reduction in the absorption time when we use RRW and specially when the metric is introduced. Also notable is the fact that while in the case of *elevud* (2,3) the absorption time is always less than the total number of states in the protocol (8), that is not the case for *elevdd* (2.3) (which also has 8 states.) As it was mentioned in Section 3, *elevdd* was built on purpose to make the "progress" towards the deadlock very slow. This characteristic influences strongly the behavior of our techniques.

To see how the absorption time into the deadlock grows with the size of the protocol in the example *elevdd*, we present the following table.

Technique	*elevdd* (2.3)	*elevdd* (2.4)	*elevdd* (2.5)	*elevdd* (2.10)
RRW	10.79	38.65	114.5	7414.95
MRWR	6.25	20.45	55.65	1945.05

Table 4.3

A close look at the results presented in Table 4.3 reveals that the growth of the absorption time in this example is exponential with the number of floors. According to this, an estimated number of 10^{200} steps would be necessary to reach the deadlock in the case of *elevdd* (3.1000) (an example with one billion states.) Although it is clear that our techniques are not likely to find the deadlock in such an example for any practical number of steps that the program is let to take, it is also true that if such a protocol were implemented in reality, it would fall into a deadlock very infrequently.

Conversely, applying MRWR to *elevud* (3.1000) (again a protocol with one billion states), results in an average absorption time of 4740.75. In this case the system

progresses rapidly towards the deadlock and our technique is able to find it very quickly.

The last set of results correspond to applying the metric aided techniques to the example *dlock(4,3,5)*. Notice that the timeout (3) is less than the maximum delay. This situation leads to the nodes timing out and retrying to acquire the locks. (As opposed to a situation in which the maximum delay were less than the timeout.) This protocol has 33,100 states. Table 4.4 contains the results. For this case, the techniques were helpful in finding the state with a number of jumps representing a small fraction of the total number of states.

Technique	*dlock* (4.3.5)
MRWU	1383.00
MRWR	987.75

Table 4.4

5. THEORETICAL BACKGROUND.

In this section we present the theory that backs up our techniques. After formalizing the random walks techniques as Markov chains, we present a way of finding bounds for the absorption time. Although these bounds are not easily generalizable, they help us understand why some methods work extremely well for particular protocols. Finally, we prove that the restricted versions of the techniques, i.e., those which do not allow the transition to the current state, always outperform their unrestricted counterparts.

As seen before, a protocol can be summarized as an edge labeled directed graph, i.e., a set of vertices G and a set of ordered pairs (x_i,x_j) of elements of G, $E(G)$, called the edges of G. In the problem under consideration G is the set of all global states and $(x_i,x_j)\in E(G)$ iff the system can evolve, given a set of decisions, from the global state x_i to the global state x_j.

From the probabilistic viewpoint, the URW model is a finite time-homogeneous Markov chain $X(n)$ whose state space is the set of all global states reachable from the initial global state and whose transition probabilities are given by

$$p(i,j)=P(X(n+1)=x_j \mid X(n)=x_i)=\begin{cases}\dfrac{1}{v(x_i)} & \text{if } (x_i,x_j)\in E(G)\\ 0 & \text{otherwise}\end{cases}$$

where $v(x_i)$ is the outdegree of x_i.

Likewise the RRW model consists of another Markov chain with the same state space and transition probabilities

$$p'(i,j)=\begin{cases}\dfrac{1}{v(x_i)} & \text{if } (x_i,x_j)\in E(G) \text{and} (x_i,x_i)\notin E(G)\\ \dfrac{1}{v(x_i)-1} & \text{if } (x_i,x_j)\in E(G) \text{and} (x_i,x_i)\in E(G)\\ 1 & \text{if } i\neq j \text{and for all } k\neq i\ (x_i,x_k)\notin E(G)\\ 0 & \textit{otherwise}\end{cases}$$

As we said before, we are interested in applying the procedures to finding a particular state in the protocols. In particular, we applied them in finding deadlock states. The following definitions characterize precisely the types of states we are looking for.

Definition 5.1 *Absorbing state.* A state $x_i \in G$ is absorbing if

$$p(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

The relationship between the URW and RRW models is given in the following lemma whose proof is a simple verification.

Lemma 5.1 If x_i is not an absorbing state then

$$p'(i,j) = \frac{p(i,j)}{1-p(i,i)}$$

otherwise

$$p'(i,j) = p(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

The MRWU and MRWR models are yet other Markov chains on the same space with transition probabilities given by

$$p''(i,j) = p_{ij},$$

where p_{ij} was defined in equations 2.2 and 2.3.

In all three models a deadlock state corresponds to an absorbing state. By basic Markov chain theory we know that if there is a path from the initial state to a set of deadlock states then eventually the chain will be absorbed with probability 1 into one of the deadlocks. This fact by itself does not provide too much information and it is of interest, of course, to know the *expected time of eventual absorption*, because we want to be assured not only that sooner or later we will find the deadlock but also that this random search will not take so long as to make preferable an exhaustive search of all the states. What follows is a brief summary of basic Markov chain theory concerning expected absorption times.

We may assume

(i) The chain is not irreducible and i is an absorbing state (= deadlock) or,

(ii) The chain is irreducible without absorbing states.

In either case we can apply the same principles corresponding to what Seneta [S81] calls ''absorbing chain techniques'': if we denote $G = \{x_1, x_2, \ldots, x_N\}$, we can relabel the states so that our interest can be expressed as finding

(i) The expected absorption time from any state x_i into the fixed absorbing state x_N, or

(ii) The expected hitting time from any state x_i to the fixed nonabsorbing state x_N.

Since the idea is the same regardless of the nature of the state x_N, we will denote in both instances by $e(x_i, x_N)$ the expected absorption (hitting) time to state x_N starting from x_i, $1 \leq i \leq N-1$.

From the transition probability matrix

$$P = (p(i,j)) \quad 1 \leq i,j \leq N$$

we delete the N-th row and N-th column to get the $N-1$ x $N-1$ matrix Q. Now the ''fundamental matrix'' $(I-Q)^{-1}$ (I is the identity) gives us all the needed information: its (i,j) entry, $(I-Q)^{-1}_{ij}$, gives the expected number of visits starting from state x_i to state x_j before absorption into (resp. hitting) the state x_N. Also, the expected time to absorption into (resp. to hit) the state x_N starting from the state x_i is given by the sum:

$$e(x_i, x_N) = \sum_{j=1}^{N-1} (I-Q)^{-1}_{ij} \tag{5.1}$$

Taking the maximum of (5.1) over all possible initial states turns out to be the inf-

norm of the fundamental matrix:

$$\max_i e(x_i, x_N) = \max_i \sum_{j=1}^{N-1} (I-Q)^{-1}_{ij} = \|(I-Q)^{-1}\|_\infty . \tag{5.2}$$

so in order to get a general bound on absorption times it would be convenient to be able to compute this norm. Of course, this computation is equivalent in difficulty to the exhaustive listing of all global states, and in general we will try to get bounds instead of exact values. For example we know that since all norm matrices are equivalent, if $\|\ \|$ is another norm for which $\|Q\|<1$, the following inequalities hold:

$$\|(I-Q)^{-1}\|_\infty \le K\|(I-Q)^{-1}\| \le \frac{K}{1-\|Q\|} \tag{5.3}$$

and we can obtain bounds for the absorption time, provided we can identify K and have some estimate for $\|Q\|$. In any case, $\|(I-Q)^{-1}\|_\infty$ provides us with a tool to justify why some methods work better than others. In particular, when we apply the above discussion to the elevator problem *elevdd* (2.3) we find that the matrix Q for URW is

$$Q = \begin{vmatrix} \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & 0 & \frac{1}{5} & \frac{1}{5} & 0 & \frac{1}{5} & 0 \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ 0 & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{vmatrix}$$

for which we get $\|(I-Q)^{-1}\|_\infty = 17.16$.

Likewise we can find the matrices Q and the values $\|(I-Q)^{-1}\|_\infty$ for all the techniques in *elevdd(2.3)* reported in Table 5.1 that shows the theoretical values and those obtained in section 4.

technique	$\|(I-Q)^{-1}\|_\infty$	Simulation
URW	17.16	16.75
RRW	13.12	16
MRWU	7.5185	6.35
MRWR	5.8889	6.25

Table 5.1

Similar computations can be carried out for *elevud(2.3)*, yielding Table 5.2, where we can compare the theoretical values to those obtained in section 4.

technique	$\|(I-Q)^{-1}\|_\infty$	Simulation
URW	6.7778	6.35
RRW	4.9444	5.7
MRWU	2.7778	3
MRWR	2.0909	2.55

Table 5.2

These computations with the inf-norm of the fundamental matrices show why we should expect MRWU to outperform RRW and this in turn to outperform URW, but they are specific to the elevator examples. At least something we can say in full generality is the following:

Theorem 5.1 RRW ouperforms URW.

Proof: Let $X(n)$ be the original URW with transition probability matrix $P = ((p(i,j))$ and let $T(0),T(1),T(2),...$ with $T(0) = 0$, be the times when the random walk jumps to a state different from the previous one (for example, if $X(0) = 3$, $X(1) = 3$, $X(2) = 3$, $X(3) = 1$, $X(4) = 1$, $X(5) = 3$, $X(6) = 7$, $\cdots$ then $T(0) = 0$, $T(1) = 3$, $T(2) = 5$, $T(3) = 6$, $\cdots$).

If we define a new Markov chain $Y(n)$ by $Y(n) = X(T(n))$ (i.e., look at the original URW only when it jumps to a new state) it turns out that this new $Y(n)$ is precisely the RRW with transition matrix $P' = (p'(i,j))$ related to the previous P by Lemma 5.1:

$$p'(i,j) = \frac{p(i,j)}{1-p(i,i)}.$$

(Roughly, the term $\frac{1}{1-p(i,i)}$ takes into account the exponential time spent in state x_i before jumping into state x_j).

Now it is clear why $Y(n)$'s absorption times are smaller than those of $X(n)$: simply by looking path-by-path: if m is the first time such that $X(m) = D$ where D is a deadlock state, then the number of different states visited prior to time m are less than or equal to m and hence $\min\{ j : Y(j) = D \} \leq m$. O

Theorem 5.2 MRWR ouperforms MRWU.

Proof: Similar to the one for Theorem 5.1. O

Finally, let us address the issue of the distinction between a protocol that does not have a deadlock and one in which the deadlock is "hard to find", i.e., the absorption time is very large. In principle, if somebody uses our techniques on a large size protocol, setting a priori a limit in the maximum number of steps of the random walk, and does not find a deadlock, he or she cannot be sure that there is no such state. (Unless the random walk had visited all the reachable states of the protocol.) Thus, in principle, we cannot distinguish between a large protocol without deadlock and a protocol that takes a long time to reach its deadlock. However, if after running our techniques for a large number of steps N, no deadlock is found, one has strong evidence that if the deadlock is

reachable, the absorption time for the protocol is bigger than *N*. Therefore, if the protocol is to be implemented, it is likely to take a long time to deadlock.

6. CONCLUSIONS

We have presented here a series of techniques that use random walks and help the designer of protocols in the task of finding particular states. These techniques have proven successful in finding deadlocks in a series of examples, although as we have seen, they cannot guarantee that if a deadlock is not found in a series of runs, it does not exist. Unless more characteristics of the protocol are known a-priori (that allowed us to use equation 5.3), we cannot give a bound on the absorption times either. Thus, if for a given protocol, after running the techniques for a fixed number of steps, we cannot find a deadlock, then our only conclusion can be that there is strong empirical evidence that the absorption time should be greater than the number of steps used. This was particularly exemplified by our custom made version of the elevator problem in which the down drift made the deadlock very hard to reach. We also have seen that for protocols that progress somewhat rapidly into deadlocks, our techniques behave very well, so as to find the state quickly.

We have proven in Section 5 that in general not letting the random walk return to the current state helps to reduce the absorption times. We also have seen that adding the metric to the search helps in general to speed up the process.

There are some other variants that we are trying to incorporate to this schema of random walk exploration. The first one exploits the idea of reducing dynamically the probabilities of the states that have been already visited, hopefully speeding up the progress towards the desired state. A difficulty with this approach is that the process thus created

is non-markovian, hence not amenable to be studied with standard Markov Chain theory.

A second idea combines the full reachability approach with the random walk techniques. It consists of stopping the random walk every so often to perform a full search of reachable states up to the *N*-th order neighbors of the current state. In this way we hope again to reduce the time needed to find the designated state.

7. REFERENCES

[ABM88] S. Aggarwal, D. Barbara, K. Z. Meth, "A Software Environment for the Specification and Analysis of Problems of Coordination and Concurrency," *IEEE Transactions on Software Engineering*,Vol. 14-3, March 1988.

[BM80] G. V. Bochmann and P. Merlin, "On the Construction of Communication Protocols," in *Proceedings of the 5th ICCC,* Atlanta, Oct. 1980.

[H78] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* , Vol.21, No. 8, Aug. 1978.

[HK89] Z. Har'El and R. Kurshan, "COSPAN: A Software System for Analysis of Coordination, " *Workshop on Automatic Verification Methods for Finite State Systems,* Grenoble, June 1989.

[K85] R.P. Kurshan, "Modeling Concurrent Processes," in *Proc. American Mathematical Society Symp. Applied Math.,* vol 31, 1985.

[MS87] N.F. Maxemchuk, and K. Sabnani, "Probabilistic Verification of Communication Protocols," in *Protocol Specification, Testing, and Verification,* North-Holland, 1987.

[P80] T.F. Piatkowski, "Remarks on the Feasibility of Validating and Testing ADCCP Protocols," in *Proc. Trends and Applications (NBS)*, Gaithersburg, 1980.

[RW83] H. Rudin and C. H. West, (eds.), *Protocol Specification, Testing, and Verification, III*, North-Holland, 1983.

[S81] E. Seneta, "Non-negative Matrices and Markov Chains," 2nd Edition, Springer-Verlag, New York 1981.

[SC88] S.M. Shatz and W.K. Cheng, "A Petri Net Framework for Automated Static Analysis of ADA Tasking Behavior," *Journal of Systems and Software*, 1988.

[W87] C.H. West, "Protocol Validation by Random State Exploration," in *Protocol Specification, Testing, and Verification*, North-Holland, 1987.