ICASE

DEC 2 6 1995

F

DISTRIBUTED STATE-SPACE GENERATION OF DISCRETE-STATE STOCHASTIC MODELS

Gianfranco Ciardo
Joshua Gluckman
David Nicol

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

19951219 014
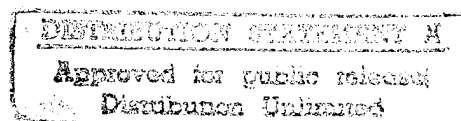
Distributed State-Space Generation of Discrete-State Stochastic Models *

Gianfranco Ciardo   Joshua Gluckman   David Nicol

Department of Computer Science

College of William and Mary

Williamsburg, VA 23187-8795, USA

{ciardo, jgluc, nicol}@cs.wm.edu

Abstract

High-level formalisms such as stochastic Petri nets can be used to model complex systems. Analysis of logical and numerical properties of these models often requires the generation and storage of the entire underlying state space. This imposes practical limitations on the types of systems which can be modeled. Because of the vast amount of memory consumed, we investigate distributed algorithms for the generation of state space graphs. The distributed construction allows us to take advantage of the combined memory readily available on a network of workstations. The key technical problem is to find effective methods for on-the-fly partitioning, so that the state space is evenly distributed among processors. In this paper we report on the implementation of a distributed state-space generator that may be linked to a number of existing system modeling tools. We discuss partitioning strategies in the context of Petri net models, and report on performance observed on a network of workstations, as well as on a distributed memory multi-computer.

---

i

1 Introduction

Discrete-state models are a valuable tool in the representation, design, and analysis of computer and communication systems, both hardware and software. We are particularly interested in stochastic formalisms, where some probability distribution is associated with the possible events in each state, so that the model implicitly defines a stochastic process. These are then used to carry on performance, reliability, or performability studies.

Most real systems, however, exhibit complex behaviors which cannot be captured by simple models having a small or regular state space. Given the high expressive power of formalisms such as Petri nets [21, 20], queuing networks, state charts [15], and ad hoc textual languages [12], the correct logical behavior can, in principle, be modeled exactly. The timing behavior is then defined by associating either an arbitrary probability distribution to the duration of each activity (resulting in a stochastic process usually solved by discrete-event simulation), or an exponential or geometric distribution (resulting in a continuous-time Markov chain (CTMC) or a discrete-time Markov chain (DTMC), respectively).

We focus on the CTMC case, where, with the exception of very special circumstances, such as the existence of product-form solutions [3, 23], or of extensive symmetries [5, 11], the numerical solution requires the generation and storage of the entire state space. This is the main drawback of the numerical approach, since the size of the state space can easily be orders of magnitude larger than what can be stored in the main memory of a single workstation.

An important aspect of the problem is that the amount of memory needed for the state-space is much larger than that needed for the numerical solution. This differential arises because, during generation, states (typically vectors of integers) must be represented whereas, during the numerical solution, integer-value codes for states suffice. Another important aspect is that the computational time spent generating a state-space is of the same order as the time spent solving it. Because of these facets, it is conceivable to generate a large model's state-space on a handful of processors, then transfer the encoded state-space to a single processor for numerical solution. This technique increases by an order of magnitude the size of models that can be solved numerically. It is an approach suitable for multiple workstations on a local-area network (LAN). The ubiquitous presence of LANs makes this approach very palatable, as it effectively offers a much larger overall amount of memory and computational power to the analyst, without requiring the purchase of new hardware. However, one should note that the serial solution phase will remain a bottleneck until parallelized. The utility of the approach is for logical analyses that can also be parallelized (we discuss instances of these), and for *capability* of solving models too large to tackle on a single workstation.

Section 2 presents the interface used to integrate an existing modeling tool with our distributed algorithm. Sections 3 and 4 discuss sequential and distributed state space explo-

ration, respectively. Section 5 presents two analysis algorithms that can be applied to the state space generated in a distributed fashion.

Our approach is not tied to a particular formalism. This greatly simplifies the parallelization of any state-space-based modeling tool. In particular, we have, for now, applied the approach to the tool SPNP [8], and report the performance results in Section 6. Section 7 summarizes our work and discusses our plans for further investigation.

2 A general interface to a distributed engine

Our goal is to provide a "distributed analysis engine" which can be connected to any "discrete-state formalism front-end". Hence, the engine implementation must not depend on the type of formalism described by the front-end. While our data are obtained by using a stochastic Petri net front-end, nothing in the engine reflects this. Indeed, we are able to integrate the engine we describe with a commercial modeling tool, BONeS Designer. Our present tool (with its capability for stationary analysis) may serve as a substitute for the transient analysis engine we've also integrated into Designer [18].

In general, we can say that the reachability set S, the set of states reachable from a given initial state s_0, is a subset of some structured countable set, often $\mathbb{N}^n$ for some n. The *reachability graph* (S, A) is a directed labeled graph whose nodes and arcs are the reachable states and the possible state-to-state transitions, respectively. Each arc is labeled with the identity e of an "event": $s \xrightarrow{e} s'$ means that event e causes a change of state from s to s'. If two events e_1 and e_2 can cause the same change of state, they correspond to distinct arcs in A. The model defines which events are enabled, i.e., can occur, in each state s, among the set of possible events E.

A model can then be considered as a way to define a set of functions which define the interface between the engine and the front-end. Besides reflecting good software engineering practice, this approach highly facilitates the integration of the engine to new front-ends. The state space S is implicitly described by the following functions:

- *Initial*, with no input parameters, returning the initial state s_0 for the model.

- *Enabled*(s), returning the (finite) set of events enabled in state $s \in S$.

- *NewState*(s, e), returning the state reached from state $s \in S$ when event $e \in Enabled(s)$ occurs.

- *Compare*(s_1, s_2), returning the result of the comparison between two states, *SMALLER*, *EQUAL*, or *LARGER*. This function prevents the engine from having to know the structure of the state, yet it allows to perform an efficient search for a given state in a

large set of states (for example using a binary search). The only assumption is that a total order can be defined over the set of reachable states.

To define the stochastic behavior, additional functions are needed, depending on the type of stochastic process underlying the model. The simplest case is when all events have exponentially distributed durations, resulting in an underlying CTMC. Then, we only need a function

- *Rate*(s, e), returning the rate at which event $e \in Enabled(s)$ occurs in state $s \in S$ in isolation.

In many models, however, it is useful to describe "instantaneous events" which can occur in zero time, as soon as they become enabled. In GSPNs [1], this is achieved by immediate transitions; in queuing networks, by passive resources. If a state enables an instantaneous event, "timed events" cannot occur, they are de facto disabled. We disallow infinite sequences of instantaneous events; while these subtle situations can be managed [6, 13], they usually indicate modeling errors. Then, the state space S can be partitioned into two classes, of "timed" and "instantaneous" states: S^T and S^I, where $s \in S^I$ iff it enables instantaneous events. The additional functions needed to describe this class of models are:

- *Timed*(s), returning *TRUE* or *FALSE* according to whether s is timed or not.

- *Weight*(s, e), returning the weight, a nonnegative real number, for the occurrence of the instantaneous event e in the (instantaneous) state s. The probability of event e in s is then obtained by normalization:

$$Prob(s, e) = \frac{Weight(s, e)}{\sum_{e' \in Enabled(s)} Weight(s, e')}$$

This definition allows for some, but not all, of the enabled instantaneous events to have zero weight in a given state s.

In certain existing tools, this interface is inadequate because it might be much more efficient to compute the rates or probabilities for all enabled events in a given state s with a single function call. We ignore this aspect for readability's sake, but we observe that the algorithms we present are not affected in any substantial way by this choice.

Finally, a model is used to study some quantity of interest. We assume this to mean the expected value of a stochastic reward process at some point in time, or in steady state. This process is defined by means of a reward rate function ρ defined over the state space: given a state $s \in S$, a given reward $\rho(s)$ is gained for each unit of time the model is in state s. Hence, for example, the expected steady-state reward rate of the model is

$$\sum_{s \in S} \rho(s)\pi_s,$$

3

where π is the steady-state probability vector. If we compute the expected cumulative time spent in each state during the interval $(t_1, t_2]$, $\sigma(t_1, t_2)$, and we use this vector instead of π, the previous sum will result in the expected cumulative reward gained during this interval, and so on. In practice, multiple reward rate functions are specified, so we define

- $Reward(s, k)$, returning the value of the k-th reward rate function evaluated in state s.

3 State space exploration

In many studies, the logical analysis of the model is of interest in itself. For example, we might want to explore qualitative properties such as absence of deadlocks and livelocks, reachability (possibility of reaching states satisfying certain conditions), liveness, and so on. If the distributions of the durations of the timed activities have unbounded support (e.g., a geometric or exponential distribution), the timing and probabilistic behavior can be ignored, except for the restriction that timed events must be considered disabled whenever an instantaneous event is enabled.

The state of the model is represented as a structured quantity, often of fixed size. For example, the state of Petri net is given by the number of tokens in each place (which could be stored as a fixed-size vector of nonnegative integers), the state of a multiclass queuing network is given by the number of customers of each class in each queue, and so on.

More complex storage schemes might be devised to save storage, often based on the existence of model invariants. For example, in a closed queuing network or in a Petri net covered by P-invariants [17], the customer populations at each queue, or the token population at each place, satisfy certain linear relationships. Sparse storage techniques can also be used to store a state, and it is possible to store an integer in just $\lceil \log k \rceil$ bits, if an upper bound k on its value is known (again invariants can be used for this purpose) [2]. We do not discuss these techniques here, since they are independent of our method and apply equally well to both sequential and distributed analysis.

Since S and (S, A) are defined only implicitly by the model, their size and characteristics might not be known a priori:

- S might be finite or infinite. We assume that it is finite, but its size is normally very large and known only at the end of the exploration.

- (S, A) might be strongly connected, or it might have a node s_i which reaches a node s_j, but is not reachable from it. We then say that node s_i is transient, since there is a positive probability that the model will never enter s_i again after leaving it.

Procedure *ExploreSequential*

1. $S \leftarrow \{Initial\}$; $S_{new} \leftarrow S$; $A \leftarrow \emptyset$;

2. while $\exists s \in S_{new}$ do

3. $S_{new} \leftarrow S_{new} \setminus \{s\}$;

4.   for each $e \in Enabled(s)$ do

5. $s_{new} \leftarrow NewState(s, e)$;

6.   if $s_{new} \notin S$ then

7. $S_{new} \leftarrow S_{new} \cup \{s_{new}\}$;

8. $S \leftarrow S \cup \{s_{new}\}$;

9.   end if;

10. $A \leftarrow A \cup \{s \xrightarrow{e} s_{new}\}$;

11.   end for;

12. end while;

Figure 1: Sequential state space exploration

In certain models, the existence of transient states is an indication of a problem, either in the model, or in the system being modeled. S contains a transient state if and only if s_0 is transient, so it is sufficient to test for this condition (see Section 5).

- Some system characteristics of interest might be studied by considering the reachability set alone. For example, we could define a complex condition which we hope never arises in the model (a circular wait in the model of a tasking system, or an inconsistent status in the model of a communication protocol). This condition can be expressed as a function $c : S \rightarrow \{0, 1\}$, where 0 corresponds to "good" states not satisfying the condition, and 1 corresponds to "bad" states. For any reachable state $s \in S$, we can then check the value of $c(s)$. Clearly, this analysis requires only to store the reachability set S (strictly speaking, it requires to enumerate it, but, if (S, A) contains cycles, the graph search requires in general to store all previously explored states, to avoid being caught into an infinite loop). Other system characteristics might require to store the reachability graph (S, A).

The sequential algorithm for state space exploration is shown in Fig. 1. If S_{new} is stored using a single-link list managed as a FIFO queue, the reachability graph is explored in breadth-first order. Each element in the list contains, either directly or by pointing to it, a different state. If A is not needed, the statements referring to it in Fig. 1 can be omitted. Note that the pseudo-code assumes that S is finite; if not, the algorithm will not halt.

3.1 Generation of the stochastic process

When the questions asked of the model refer to the timing and stochastic behavior, a stochastic process (not just a reachability graph) must be built as a result of the state space exploration. If the underlying process is a CTMC, this means building an infinitesimal generator matrix Q, where $Q_{s,s'}$ is the rate of going from state $s \in S^T$ to state $s' \in S^T$, for $s \neq s'$. The diagonal entries of Q are defined to be the negative of the sum of the off-diagonal entries on the corresponding rows, $Q_{s,s} = -\sum_{s' \in S^T, s' \neq s} Q_{s,s'}$; in our discussion, we assume that they are stored explicitly only during the CTMC solution. It is usually more efficient to use an alternative exploration algorithm, which stores only the timed states T. Analogously, only a "reduced reachability graph", basically equivalent to Q, needs to be stored. Whenever an instantaneous state is found, a depth-first search is initiated, to determine the set of timed states reached. If

$$s \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots \xrightarrow{e_{k-1}} s_k \xrightarrow{e_k} s'$$

where $s, s' \in S^T$ and $s_1, s_2, \ldots, s_k \in S^I$, the rate of going from s to s' along this path is

$$Rate(s, e_0) \cdot \prod_{i=1}^{k} Prob(s_i, e_i)$$

and $Q_{s,s'}$ is the sum of these rates over all possible paths of this type from s to s', including paths of length one, that is, paths with no intermediate immediate states.

This is illustrated in Fig. 2. Procedure $BuildQ$ is called first, which in turn uses the recursive procedure $GenerateTransitions$. For simplicity, we assume that S^T, S^T_{new}, S^I_{new}, and Q are global variables and that the initial state is timed (the algorithm can be easily modified if this is not the case). S^I_{new} is used as a stack, and is needed to recognize cycles of instantaneous events, which we consider illegal. When the execution returns to $BuildQ$, S^I_{new} is empty, that is, instantaneous states are stored only temporarily. Note that entries of Q are incremented, not set, by statement 6 in procedure $GenerateTransitions$. This is because multiple paths of instantaneous states might exist between the same source and destination.

4 Distributed state space exploration

Like the sequential algorithm, the distributed algorithm shown in Fig. 3 performs a breadth-first exploration of the state space. Each state reached is either explored locally or sent to another process. For the distributed algorithm, we then define another function in the interface:

- $Partition(s, N)$, returning the identity of the process to which state s is assigned, an integer between 0 and $N - 1$.

Procedure *BuildQ*

1. $S^T \leftarrow \{Initial\}$; $S^T_{new} \leftarrow \{Initial\}$; $S^I_{new} \leftarrow \emptyset$; $Q \leftarrow 0$;

2. while $\exists s \in S^T_{new}$ do

3. $S^T_{new} \leftarrow S^T_{new} \setminus \{s\}$;

4.   for each $e \in Enabled(s)$ do

5. $s_{new} \leftarrow NewState(s, e)$;

6. $GenerateTransitions(s, Rate(s, e), s_{new})$;

7.   end for;

8. end while;

Procedure *GenerateTransitions(t, r, t_{new})*

1. if $Timed(t_{new})$ then

2.   if $t_{new} \notin S^T$ then

3. $S^T_{new} \leftarrow S^T_{new} \cup \{t_{new}\}$;

4. $S^T \leftarrow S^T \cup \{t_{new}\}$;

5.   end if;

6. $Q_{t,t_{new}} \leftarrow Q_{t,t_{new}} + r$;

7. elsif $t_{new} \notin S^I_{new}$ then

8. $S^I_{new} \leftarrow S^I_{new} \cup \{t_{new}\}$;

9.   for each $e \in Enabled(t_{new})$ do

10. $GenerateTransitions(t, r \cdot Prob(t_{new}, e), NewState(t_{new}, e))$;

11.   end for;

12. $S^I_{new} \leftarrow S^I_{new} \setminus \{t_{new}\}$;

13. else

14. $error(\text{``cycle of instantaneous events''})$;

15. end if;

Figure 2: Sequential generation of Q

Assuming we have N processes running on N processors, this function partitions the state space into N classes, one assigned to each process(or), and is a critical factor affecting the performance of the distributed algorithm (see Section 6.1).

The incidence matrix of the reachability graph is stored in column-wise format. Hence, when process i determines that state s_{new} "belongs" to a remote process $j \neq i$, it sends both s_{new} and the arc leading to it, $s \xrightarrow{e} s_{new}$, to j. As the representation of a state can be quite

7

Procedure $ExploreDistributed_i$

1. if $Partition(Initial) = i$ then
2. $S^i \leftarrow \{Initial\}$;
3. else
4. $S^i \leftarrow \emptyset$;
5. end if;
6. $S^i_{new} \leftarrow S^i$; $A^i \leftarrow \emptyset$;
7. while "*not received terminate message*" do
8.   while $\exists s \in S^i_{new}$ do
9. $S^i_{new} \leftarrow S^i_{new} \setminus \{s\}$;
10.   for each $e \in Enabled(s)$ do
11. $s_{new} \leftarrow NewState(s, e)$;
12. $j = Partition(s_{new}, N)$;
13.   if $j \neq i$ then
14. $SendState(j, s_{new})$;
15. $SendArc(j, s \xrightarrow{e} s_{new})$;
16.   else
17.   if $s_{new} \notin S^i$ then
18. $S^i_{new} \leftarrow S^i_{new} \cup \{s_{new}\}$;
19. $S^i \leftarrow S^i \cup \{s_{new}\}$;
20.   end if;
21. $A^i \leftarrow A^i \cup \{s \xrightarrow{e} s_{new}\}$;
22.   end if;
23.   end for;
24.   end while;
25. $S^i_{new} \leftarrow S^i_{new} \cup ReceiveStates$;
26. $A^i \leftarrow A^i \cup ReceiveArcs$;
27. end while;

Figure 3: Distributed state space exploration using N processes

large, states are assigned an index. Locally (i.e., in process i), state $s \in S^i$ is identified by the index k, if s is the k-th state added to S^i. Globally (i.e., in process $j \neq i$), state s is identified by a two-component index: (k, i). State indices, rather than actual states, are stored and exchanged whenever possible.

When i sends state s_{new} to j, with the function call $SendState(j, s_{new})$, the local index of s_{new} in j is not known to i, so the actual state must be sent. However, when the arc $s \xrightarrow{e} s_{new}$ is sent, with the function call $SendArc(j, s \xrightarrow{e} s_{new})$, the global state index of s is sent, since it is known to i. Also, the destination (actual) state s_{new} does not have to be sent a second time to describe the arc, since states and arcs are always sent in pairs.

The functions $ReceiveStates$ and $ReceiveArcs$ return all the states and arcs sent to process i since the last time they were called, respectively. Each state x in the set returned by $ReceiveStates$ has a corresponding arc $(k_1, j) \xrightarrow{e} x$ in the set returned by $ReceiveArcs$. The set S^i is then searched. If $x \in S^i$, its previously assigned index is retrieved, otherwise x is added to S^i and a new index is generated for x. If k_2 is the local index of x, an arc $(k_1, j) \xrightarrow{e} (k_2, i)$ is added to A^i. These are high-level descriptions; the actual implementation details for the communication mechanism are beyond the scope of this presentation.

In the sequential algorithm, the choice between storing the incidence matrix of the reachability graph in row-wise or column-wise format is irrelevant. For the distributed version, however, a row-wise format would require a more complex protocol. With row-wise storage, the entry $s_1 \xrightarrow{e} s_2$ is stored by process i as $(k_1, i) \xrightarrow{e} (k_2, j)$, if s_1 is assigned index (k_1, i) and s_2 is assigned index (k_2, j). However, i does not know the local index k_2 of s_2, so it must send s_2 to j, and wait for (k_2, j) in return. Only then i can complete the storage of the entry in the incidence matrix. With the column-wise format we use, i simply sends the pair s_2 and $(k_1, i) \xrightarrow{e} s_2$ to j, without having to wait for further information from j, since it is up to j to fill-in the value for the arc destination.

The communication complexity of the distributed state-space algorithm is then one (actual) state, one state index, and one event, for each "cross-arc" (an arc from a state in S^i to a state in S^j, $i \neq j$).

When process i finishes exploring its local states (S^i_{new} is empty), it waits for more states and arcs from other processes. When all processes have finished their local work and are waiting to receive a message, the distributed state space exploration has completed. Detecting termination is a well-known problem with many solutions. In the workstation network data we present, we used the circulating probe algorithm described by Dijkstra et al. [10]. We have since made the engine portable by using MPI [14] as the communication mechanism, and in that context employ the scalable "Non-committal barrier" described by Nicol [19].

4.1 Distributed generation of the stochastic process

For brevity's sake, we do not present the pseudo-code for the distributed generation of the underlying stochastic process, obtained by merging the algorithm for the distributed generation of the state space of Fig. 3 with the elimination of the immediate states used in

the algorithm of Fig. 2. Only timed states are assigned to a particular process using the partition function. Immediate states are managed in the process that generates them, and then discarded after all the timed states reachable from them have been explored. Storing the immediate states together with the timed ones is a reasonable alternative (see [6, 4] for the tradeoffs involved in storing these states permanently), but is probably less appropriate if the paramount goal is to minimize storage requirements.

At the end, process i contains the states $S^{T,i} = \{s \in S^T : Partition(s, N) = i\}$ and the entries of Q corresponding to arcs reaching these states, $Q_{\bullet, S^{T,i}}$.

4.2 Implementation issues

Before concluding this section, we discuss a few implementation issues. Communication between processes is accomplished through message passing. Reliable message passing is provided by acknowledged messages: a sender does not continue until the receipt of its message has been acknowledged. Since the receipt of a message generates a signal, the receiver can acknowledge the message almost immediately, thus minimizing the waiting time for the sender.

Because of the potentially high number of states sent to another process, each state/arc pair is buffered in the sender. The buffer size is a compilation parameter. As the size increases, more states and arcs can fit into a single message, and fewer, although larger, messages are exchanged. This reduces one type of overhead, but it also increases the likelihood that a process j remains idle waiting for states to be imported, while some other process i delays sending states that j should explore because the buffer is not full.

5 Distributed analysis of the model

Once the state space is built, analysis can proceed. We present two types of state-space-based analysis. The first type analyzes logical properties of the state-space; we point out a number of questions that can be answered in a distributed fashion using the distributed state-space. The second type is numerical analysis; we presently perform this sequentially, but point out some unexpected ramifications of our distributed generation of the state-space. In the following, we use the following symbols:

- n and η are the total number of tangible states and arcs stored; $n = |S^T|$ is the dimension of Q and η is the number of nonzero entries in Q: $\eta = |\{(s_1, s_2) : Q_{s_1,s_2} > 0\}|$.

- n^i and η^i are the analogous quantities for process i in the distributed algorithm: $n^i = |S^{T,i}|$ and $\eta^i = |\{(s_1, s_2) : Q_{s_1,s_2} > 0, s_2 \in S^{T,i}\}|$. Furthermore, we define $\eta^{j,i}$ to be the number of entries stored by i corresponding to events originating from states in j:

$\eta^{j,i} = |\{(s_1, s_2) : Q_{s_1,s_2} > 0, s_1 \in S^{T,j} \wedge s_2 \in S^{T,i}\}|$. Hence, $\sum_{i=0}^{N-1} n^i = n$, $\sum_{i=0}^{N-1} \eta^i = \eta$, and $\sum_{j=0}^{N-1} \eta^{j,i} = \eta^i$.

5.1 Distributed detection of logical state properties

The notion of reachability pervades logical analysis of state-spaces: "is it possible to reach some state s_1 from another state s_0?". As we will see, solution to this problem permits one to address higher-level questions. For instance, to determine whether there are any transient states, it is sufficient to test whether the initial state is transient, that is, whether there exists a state $s \in S^T$ which does not reach s_0. This is equivalent to determining whether there is a state s unreachable from s_0 in the "reverse reachability graph", obtained by reversing the direction of all arcs. A simple breadth-first search algorithm that identifies all states reachable from s_0 in the reverse graph can be used for this purpose—if any state remains untouched, s_0 is transient. An efficient implementation requires a row-wise storage of the incidence matrix of the reverse graph, and this is a further reason to use a column-wise format for the storage of the incidence matrix of the original graph, since one is the transpose of the other. Note that the state-space generation process is itself a breadth-first search, and that the algorithm for testing whether s_0 is transient is essentially the same.

A sequential breadth-first search algorithm requires $O(\eta)$ operations. A distributed implementation requires the same number of operations, but also $O(\chi)$ communication, where

$$\chi = \sum_{i,j \in \{0,...,N-1\}, i \neq j} \eta^{j,i}$$

is the number of cross-arcs.

The communication cost is then of the same order of complexity as for the state space generation, although now only state indices, not the actual states, are sent between processes.

Many other important questions about the behavior of a system can be answered in a distributed way using the information in the reachability set and graph:

- Reachability: a condition c is reachable if there is a state $s \in S$ satisfying c. Each process i can simply test for this every time it adds a new state to S^i, that is, this question can be answered without further examining the reachability graph. Deadlocks are just a special case: a deadlock is an absorbing state, that is, a state which does not enable any event.

- Livelock: a livelock is a set of states L, $1 < |L| < |S|$ such that, once L is entered, no state in $L - S$ can be reached. Formally, the reachability graph must contain a strongly connected component with two or more nodes and no outgoing arcs (no way to leave the component). This is equivalent to testing whether the initial state is transient

in a modified reachability graph without absorbing states (these can be easily tagged during state-space generation).

- Liveness: an event e is not live if there is a state $s \in S$ such that, once s is entered, e can never become enabled. If we define S_e to be the set of states where e is enabled, liveness can be established by checking that every state in S can reach a state in S_e. The same breadth-first algorithm used to determine whether the initial state is transient can be adopted; the only difference is that the search in the reverse graph proceeds from the set of states S_e, not from s_0 alone.

- Conditional reachability: we are sometimes interested in determining whether there exist two states s_1, satisfying condition c_1, and s_2 satisfying condition c_2, such that s_1 reaches s_2. If the entire graph is strongly connected, this is equivalent to asking whether conditions c_1 and c_2 can be satisfied. Otherwise, a modified version of distributed breadth-first search algorithm used to determine whether s_0 is transient can be employed. Instead of starting from s_0, we start from a set of states $S_2 = \{s : c_2$ is satisfied in $s\}$, and we determine the set of states R reachable from S_2 in the reverse graph. Our goal is to find a state in R satisfying s_1.

5.2   Numerical solution of the underlying stochastic process

In the current implementation, the numerical solution of the CTMC is centralized. While this prevents us from obtaining good speedups, it is important to remember that our immediate goal is to increase the size of models we can solve. As pointed out earlier, the large difference in memory requirements for the generation and solution phases means we can solve models on one processor that are an order of magnitude larger than we can generate on one processor.

After building Q and testing for a transient initial state, each process sends its portion of Q to a solver process where the numerical solution is performed. This is reasonable given our current target level of parallelism, up to a dozen workstations.

The "solution" sought for the CTMC is normally the steady-state probability vector π satisfying $\pi Q = 0$, if Q is ergodic, or the sojourn times in each transient state until absorption, or the transient instantaneous or cumulative probability in each state. In any case, the solution is given by a real vector v of size n.

Vector v is generally not all that one wishes to know about the model. Rather, v is used to compute the expected "reward" earned by the model. Rewards are a function of individual states; to compute the reward for state s one must generally have available the full state representation of s. Consequently, after computing v serially, we distribute it back to the processors holding the full state-space description. Recall now that the compact representation of a state identifies the processor that owns it. It is straightforward then to

return each component v^i to process i, for $i \in \{0, \ldots, N-1\}$. It is then possible to compute the expected value of a measure, $m = \sum_{s \in S^T} Reward(s) \cdot v_s$ in a distributed way. Process i computes $m^i = \sum_{s \in S^{T,i}} Reward(s) \cdot v_s$, and a master process combines these subresults as $m = \sum_{i=0}^{N-1} m^i$.

Note that a number of measures may be computed simultaneously simply by using different reward functions. Since the number of requested measures can be quite large in practice, their distributed computation can result in a substantial time saving.

An interesting observation should be made at this point. It is well known that the ordering of the variables (states) can affect the speed of convergence for iterative methods such as Gauss-Seidel and Successive-Over-Relaxation (SOR) [24, 22]. We indeed experienced this phenomenon when studying the number of iterations required by a sequential SOR implementation. In our first implementation, all the states in $S^{T,i}$ were ordered before those in $S^{T,i+1}$, $i \in \{0, 1, \ldots, N-1\}$, while the sequential state-space exploration results in a breadth-first order, starting from s_0. The ordering from the distributed implementation regularly required more iterations, even if the SOR implementation was exactly the same.

We conclude that the natural breadth-first order by which states are generated and indexed in the sequential implementation is a better choice. To verify this, we sorted the states in the solver process according to a breadth-first order: if the distance from s_0 to s_i is less than the distance from s_0 to s_j, s_i is assigned an index smaller than s_j. This does not necessarily achieve exactly the same order as in the sequential implementation (since multiple total orders are compatible with the above partial order), but it does result in approximately the same number of iterations in the two implementations. We believe that state ordering will become an issue in a distributed implementation, where it requires reshuffling states, and the corresponding columns of Q, among the N processes.

The partition heuristic might affect the convergence of a distributed solution in other ways as well. We have not yet considered these aspects in detail, but it is clear that the actual numerical values of the rates of the state-to-state transitions, rather than the mere existence or absence of an arc, will need to be taken into account in this case.

For example, the idea of decomposability [9] is based on finding a block partition of the transition matrix where the entries of the off-diagonal blocks are orders of magnitude smaller than those in the diagonal blocks. This ensures that, after entering a block, the stochastic process reaches an "approximate steady-state" before moving to a different block. If the partition heuristic is such that each class corresponds to one or more blocks having this property, then several attractive iterative methods will be appropriate, since most of the iterations will occur within a single class, while only a few global iterations requiring the exchange of data across processors will be needed. A good candidate for this application is the multi-level algorithm developed by Horton and Leutenegger for the numerical solution
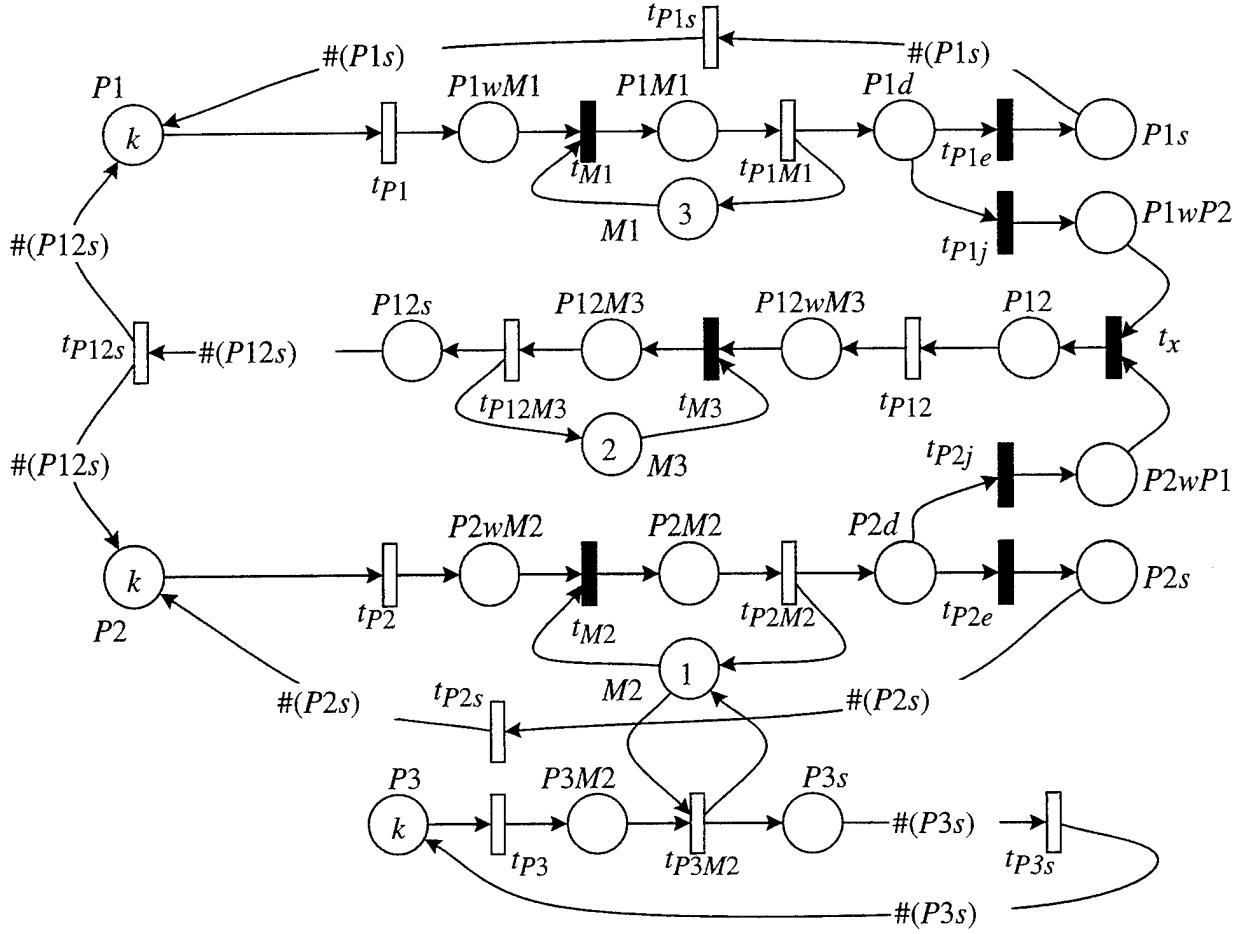
Figure 4: The FMS stochastic Petri net.

of Markov chains [16].

6 Results

We consider the model of a flexible manufacturing system (FMS) shown in Fig. 4. We omit a description of this SPN, since we are focusing on a comparison of the sequential and distributed algorithms for its analysis. The interested reader can consult [7] for a detailed presentation of its behavior and the meaning of its places and transitions. For this discussion, it is sufficient to observe that, as the number k of initial tokens in the three places $P1$, $P2$, and $P3$ increases, the number of states n and arcs η increases sharply (see Table 1). The first partitioning function used (to be described) assigns states to processors depending on the markings in places $P1$, $P2$, $P3$.

The largest size listed ($k = 6$) exceeded the storage capacity of a single workstation, yet was solvable (in approximately 40 minutes) by generating the state-space using five
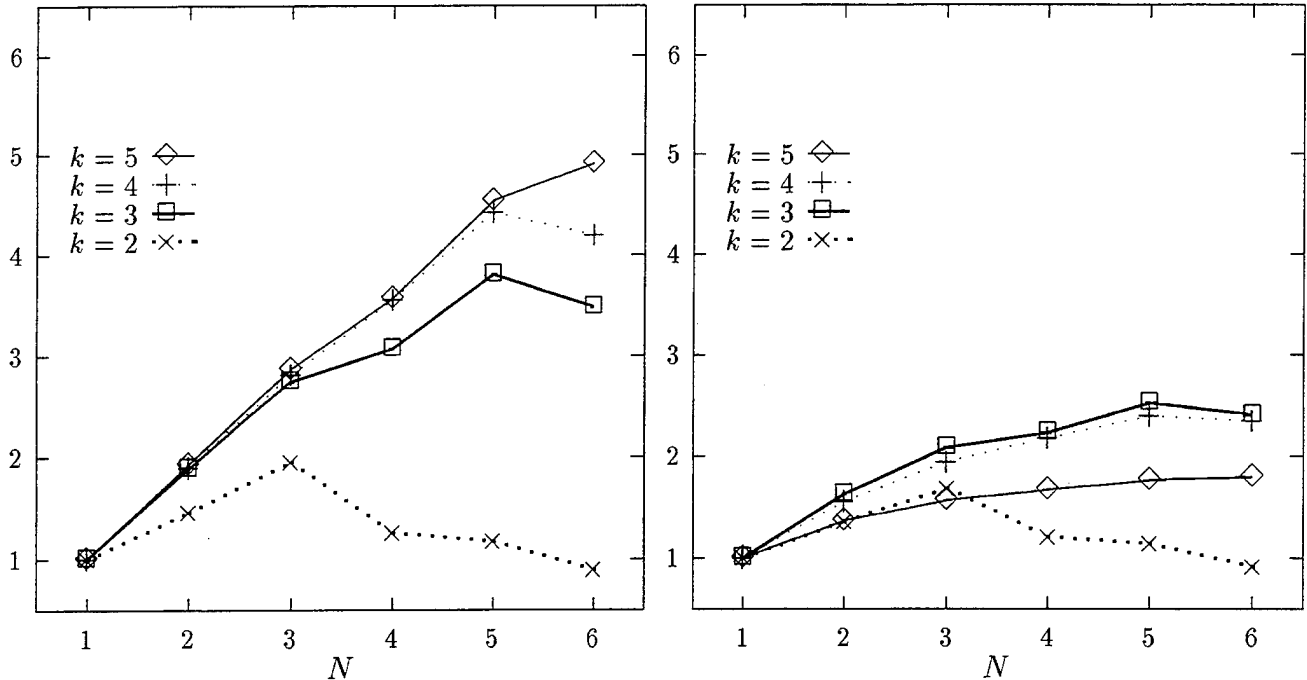
Figure 5: Speedup for generation alone (left) and for overall solution (right).

processors, and solving it on one. To assess how well the generation process parallelizes, we consider speedup on problems small enough to solve on one processor. Fig. 5 shows the speedup obtained by running our distributed algorithm on a set of homogeneous workstations (SPARCstation 10 class) communicating over a 10Mbs Ethernet, for various values of the initial number of tokens k. The left plots refer to the timing collected for the generation of the state space alone, while the right plots refer to the timing for the entire solution process, assuming a single measure: the expected total number of tokens in $P1$, $P2$, or $P3$, in steady state. As previously noted, computing a single measure results in the lowest possible overall speedup since the computation of the measures can be almost perfectly parallelized.

The speedup with N processes is obtained by dividing the runtime of the sequential solution ($N = 1$) by the runtime of the distributed solution with N processes (defined as the maximum processor runtime). The runtime of a process includes the time spent executing user or system instructions and the time spent communicating or waiting for states to be imported, if $N > 1$. Speedups are calculated only for model problems where the serial solution ran without paging (other than to load, of course). Speedup of the generation phase thus measures the relative cost of the communication overhead during that phase.

Since our motivation is exploiting distributed memory, the case of highest interest is that of $k = 5$, the largest problem. Here we see evidence of a favorable computation to communication balance (as well as good load balance), since speedup increases almost linearly in the number of processors. The complete serial solution required nearly an hour of computation.
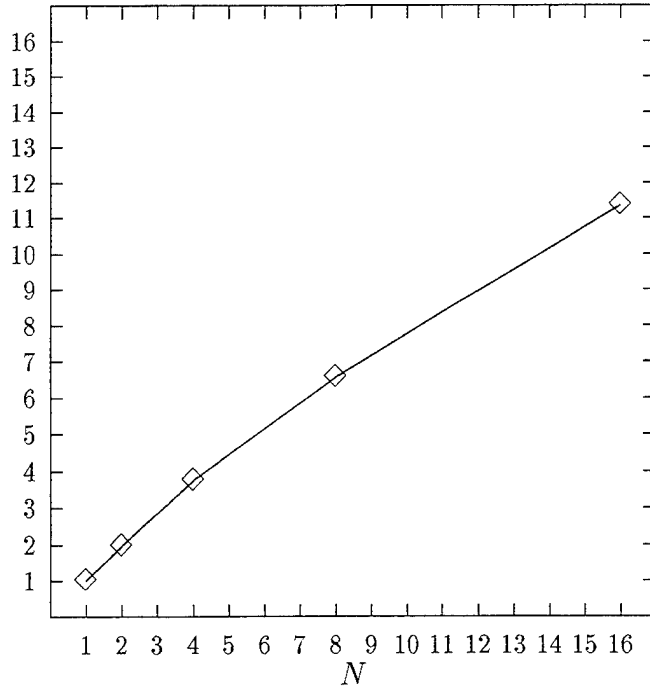
Figure 6: Speedup on the SP-2 ($k = 5$).

k	n	η
1	54	155
2	810	3,699
3	6,520	37,394
4	35,910	237,120
5	152,712	1,111,482
6	537,768	4,205,670

Table 1: Size of the tangible reachability set and graph as a function of k.

We also ported our distributed engine to an IBM SP-2 multiprocessor. Fig. 6 shows the speedup for the state-space generation on the same FMS SPN, for the case $k = 5$, as a function of N. We achieved a speedup of 11.35 when $N = 16$. The serial state-space generation requires 14 minutes on a single processor. We also experimented with larger values of k, and various machine sizes. With $k = 6$ there are $537,768$ states. The time needed to generate the state-space varied from three hours using 2 SP-2 processors (the smallest configuration able to solve the problem), to 18 minutes using 32 processors. Using 32 processors we were able to generate the $k = 7$ state-space ($1,639,440$ states) in 51 minutes, and the $k = 8$ state-space ($4,459,455$ states) in four hours.

The conclusion we may draw from this data is that the algorithm works well, and makes
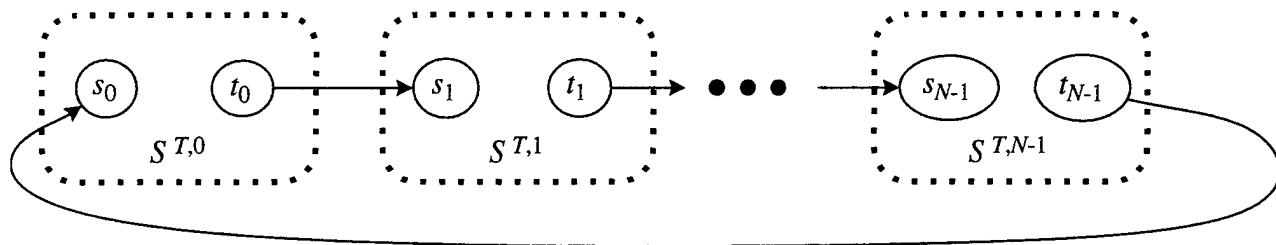
Figure 7: An extreme case for the distributed algorithm.

possible the generation of state-spaces that are much larger than those usually considered tractable.

6.1 Choosing a good partition function

Choice of a good partition function is critical. It must provide both *locality* (if possible), and *balance*. Locality means that, in general, most of a state's descendents are assigned to the same processor as is the parent state. Locality reduces communication overhead. Spatial balance means that each processor is assigned approximately the same number of states; contrast this with temporal balance which additionally calls for each processor to be busy most of the time. Spatial balance is sufficient if problem-solving capacity is a concern, whereas temporal balance is required to achieve good speedups.

The modeling paradigm may provide clues to locality. The SPN reported in this paper is a good example. A Petri net state is a vector enumerating the tokens in each place. When a transition fires, typically only a small number of components of this vector change. Thus, if we base a partitioning rule on the markings of a small fixed set of places (called a *control set*), all transition firings that do not involve control set places reflect state transitions that are contained entirely within a processor. To achieve spatial balance we first need to choose the control set so that the range of combinations of markings in its places is large. Unfortunately, in the worst case one needs to generate the state-space to discover just what that range is; we must therefore rely upon the user's intuition about the model to provide this property. Lastly, given a wide spread of markings in the control set, we assign a state to a processor by applying a hashing function to the marking of its control set.

To further illustration the difference between spatial and temporal balance, consider the extreme case of Fig. 7, where only one arc, $t_i \rightarrow s_{(i+1) \bmod N}$, connects $S^{T,i}$ to $S^{T,(i+1) \bmod N}$, and $s_0 \in S^{T,0}$. If state t_i is the last one examined in each $S^{T,i}$, the distributed algorithm will run sequentially, even if the states might be evenly allocated onto the N processes, and the number of cross arcs is certainly minimum: $\chi = 6$. On the other hand, even in this unfortunate situation, the distributed algorithm would still have an advantage over the

17

sequential one, since communication overhead would be negligible, and the entire amount of memory available on N processors would be available.

For the problem whose performance we studied, $\{P1, P2, P3\}$ is the control set. The hashing function is

$$(\#_{P1} + q \cdot \#_{P2} + q^2 \cdot \#_{P3}) \bmod N$$

where q is a prime number (1013, in our case) and $\#_p$ indicates the number of tokens in place p (in a given marking). The bar chart on the top right in Fig. 8 shows the distribution of states using this partitioning function with $N = 6$ processes.

A good tool to decide the quality of the partition function is then the matrix of the numbers of edges cut by the partition, $\eta^{i,j}$. Fig. 8 describes these values for four different choices of the partition function. The parameters considered are six processes ($N = 6$) and five tokens initially in $P1$, $P2$, and $P3$ ($k = 5$), resulting in 152,712 states and 1,111,482 arcs in the timed-to-timed state-space and graph, respectively. These values are obviously independent of the partition function chosen.

A simple hashing on a few components of the state description achieves a reasonably uniform allocation of states to processes. Using *all* the components might not be a good idea for several reasons:

- If linear invariants exist relating the values of the components, and if these values are simply summed, the partition function might not achieve a good "random" spread. For example, in a closed queuing network, the choice "sum of the number of customers in each queue mod N" would allocate all the states to the same process, a bad choice. Petri nets also often exhibit this type of invariants.

- Multiplying the components by some power of a large prime number, as we did, eliminates most problems due to the existence of invariants. However, in this case, if every component is factored in the computation of the partition function, most, if not all, arcs will be cross-arcs, because every event changes one or more components of the state.

Hence, it is best to use only a few components of the state in the definition of the partition function. Any event which changes only the values of the other components is then guaranteed not to generate any cross-arcs (e.g., t_{P12M3} in Fig. 4, with the first partition function).

If a particular structure is desired for the partition of Q, an appropriate partition function might be employed. The third choice in Fig. 8,

$$(\#_{P1wM1} + \#_{P1M1} + \#_{P2wM2} + \#_{P2M2} + \#_{P3M2}) \bmod N$$

for example, results in a block-tridiagonal structure for Q, except for two blocks, in the upper-right and lower-left corners, due to the wrap-around nature of the modulo operator. This happens because we intentionally chose the control set so that any events can change at most one control set place marking, by at most value 1. Such a sparsity pattern might be very desirable, depending on the type of communication available between the workstations. If they were connected in a circular fashion with bidirectional links, they could potentially be all communicating at the same time, since each workstation only needs to exchange data with its two neighbors.

We observe that the difference between the first and second partition function in Fig. 8 is just in the multiplication by powers of 1013 in the first case, resulting in a 10% reduction in the number of cross arcs, while the state distribution is substantially similar. The price paid to obtain the tridiagonal structure with the third partition function is instead a higher number of cross arcs (20% more than for the first partition function). Interestingly enough, the second and third partition functions result in exactly the same state distribution. This is due to the existence of invariants, ensuring, for example, that the number of states where $\#_{P1} = m$ or $\#_{P1wM1} + \#_{P1M1} = m$ is exactly the same, and so on. Finally, the fourth function minimizes the number of cross arcs (fewer than a quarter of the arcs are cross arcs), and also achieves a good distribution of states. The clearly recognizable patterns in the matrix and in the state distributions are due to the independence of the rest of SPN from the number of tokens in $P3$ and $P3M2$. There are $\binom{5+3-1}{5} = 21$ ways to distribute $k = 5$ tokens over three places. For every such combination, exactly 7,272 combinations of tokens in the other places exist ($7,272 \times 21 = 152,712$, the total number of states). Hence, the value of $|S^{T,i}|$ is easily determined once we know how many of the 21 combinations correspond to process i. The hashing enforced by the expression $(\#_{P3} + \#_{P3M2} \cdot 1013) \bmod N$ results in three combinations assigned to processes 0, 2, and 4 ($|S^{T,0}| = |S^{T,2}| = |S^{T,4}| = 3 \times 7,272 = 21,816$), and four combinations assigned to processes 1, 3, and 5 ($|S^{T,1}| = |S^{T,3}| = |S^{T,5}| = 4 \times 7,272 = 29,088$).

Clearly, much work remains to be done in this area, but the good news is that even just moderately informed choices, such as the first two in Fig. 8 still achieve our goals of locality, spatial balance, and temporal balance.

7 Conclusion and future work

We have demonstrated the feasibility of distributing the state-space generation phase of discrete state stochastic system analysis using only a small network of workstations. The approach exploits the memories of multiple workstations, allowing one to build and perform logical analyses of state-spaces too large for a single processor. We stress that our approach provides a distributed algorithm which is independent of the particular user-level formalism

adopted for the specification of the model.

To improve the applicability and usefulness of our approach, two aspects need to explored further. First, a centralized numerical solution is appropriate when using up to a dozen or so workstations, or when a machine particularly suited for numerical computation and equipped with a substantial amount of memory is available. However, our approach is a natural candidate for a completely distributed implementation, so we intend to explore the rich area of distributed solutions of linear systems, and implement some of the most appropriate techniques. This becomes a necessity if we hope to scale up to a much larger number of workstations.

Second, the efficiency of our approach is highly sensitive to the partition heuristics used. Hence, we plan to investigate algorithms to derive a "good" partition from an automatic structural analysis of the model, that is, before starting to generate the state-space. This is doubly important because the specification of the heuristics, in addition to being a critical factor, is a new burden put upon the user with respect to the sequential solution.

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comp. Syst.*, 2(2):93–122, May 1984.

[2] G. Balbo, G. Chiola, G. Franceschinis, and G. Molinari Roet. On the efficient construction of the tangible reachability graph of generalized stochastic Petri nets. In *Proc. 2nd Int. Workshop on Petri Nets and Performance Models (PNPM'87)*, Madison, Wisconsin, Aug. 1987. IEEE Computer Society Press.

[3] F. Baskett, K. M. Chandy, R. R. Muntz, and F. Palacios-Gomez. Open, Closed, and Mixed networks of queues with different classes of customers. *Journal of ACM*, 22(2):335–381, Apr. 1975.

[4] A. Blakemore. The cost of eliminating vanishing markings form generalized stochastic Petri nets. In *Proc. 3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 85–92, Kyoto, Japan, Dec. 1989. IEEE Computer Society Press.

[5] P. Buchholz. Hierarchical Markovian models – Symmetries and Reduction. In *Modelling Techniques and Tools for Computer Performance Evaluation*. Elsevier Science Publishers B.V. (North-Holland), 1992.

[6] G. Ciardo, J. Muppala, and K. S. Trivedi. On the solution of GSPN reward models. *Perf. Eval.*, 12(4):237–253, 1991.

[7] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.

[8] G. Ciardo, K. S. Trivedi, and J. Muppala. SPNP: stochastic Petri net package. In *Proc. 3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 142–151, Kyoto, Japan, Dec. 1989. IEEE Computer Society Press.

[9] P. J. Courtois. *Decomposability: Queueing and Computer System Applications*. Academic Press, New York, NY, 1977.

[10] E. W. Dijkstra, W. Feijen, and A. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Inf. Proc. Letters*, 16:217–219, June 1983.

[11] S. Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets amenable to parallel solution. In *Proc. 4th Int. Workshop on Petri Nets and Performance Models (PNPM'91)*, Melbourne, Australia, Dec. 1991. IEEE Computer Society Press.

[12] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi. The System Availability Estimator. In *Proc. 16th Int. Symp. on Fault-Tolerant Computing*, pages 84–89, Vienna, Austria, July 1986. CS Press.

[13] W. K. Grassmann and Y. Wang. Immediate events in Markov chains. In W. J. Stewart, editor, *Numerical Solution of Markov Chains '95*, pages 163–176, Raleigh, NC, Jan. 1995.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, Cambridge, Mass, 1994.

[15] D. Harel. On visual formalisms. *Communications of ACM*, 31(5), May 1988.

[16] G. Horton and S. T. Leutenegger. A multi-level solution algorithm for steady state Markov chains. In *Proc. 1994 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 191–200, Nashville, TN, May 1994.

[17] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalised Petri net. In *Proc. 2nd European Workshop on Application and Theory of Petri Nets*, pages 411–422, Bad Honnef, Germany, 1981.

[18] D. Nicol, D. Palumbo, and M. Ulrey. A graphical tool for reliability and failure-mode-effects analysis. In *Proc. of the 1995 Reliability and Maintainability Conference*, pages 74–81, Jan. 1995.

[19] D. M. Nicol. Non-commital barrier synchronization. *Parallel Computing*, 21:529–549, 1995.

[20] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[21] C. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, Bonn, West Germany, 1962.

[22] S. Pissanetzky. *Sparse Matrix Technology.* Academic Press, 1984.

[23] M. Sereno and G. Balbo. Computational algorithms for product form solution stochastic Petri nets. In *Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM'93)*, pages 98–107, Toulouse, France, Oct. 1993. IEEE Computer Society Press.

[24] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.

	0	1	2	3	4	5
0	104,265	31,123	10,844	9,345	9,376	38,528
1	47,925	107,937	30,878	11,797	10,062	9,588
2	7,640	47,830	97,875	28,272	11,235	9,027
3	6,729	6,103	42,542	83,385	25,288	9,698
4	7,750	5,694	5,541	36,516	73,881	24,239
5	27,516	7,962	6,411	6,494	33,967	78,219

$(\#_{P1} + \#_{P2} \cdot 1013 + \#_{P3} \cdot 1013^2) \bmod N$

$\chi = 565,920$ (50.9% of η)

	0	1	2	3	4	5
0	73,269	18,070	16,619	6,427	5,065	37,889
1	43,422	87,870	20,373	19,878	8,433	7,345
2	7,671	51,855	96,147	22,236	22,286	10,052
3	9,413	6,723	56,546	94,080	22,644	22,731
4	19,805	7,587	5,150	55,090	82,395	21,368
5	17,349	15,108	5,356	3,464	47,782	63,984

$(\#_{P1} + \#_{P2} + \#_{P3}) \bmod N$

$\chi = 613,737$ (55.2% of η)

	0	1	2	3	4	5
0	63,282	41,130	0	0	0	48,253
1	59,677	76,956	49,782	0	0	0
2	0	71,701	84,660	54,948	0	0
3	0	0	77,209	82,662	54,012	0
4	0	0	0	74,185	71,694	47,148
5	35,940	0	0	0	63,715	54,528

$(\#_{P1wM1} + \#_{P1M1} + \#_{P2wM2} + \#_{P2M2} + \#_{P3M2}) \bmod N$

$\chi = 677,700$ (61.0% of η)

	0	1	2	3	4	5
0	120,906	13,536	0	7,272	14,544	7,272
1	0	161,208	9,396	7,272	0	29,088
2	14,544	0	120,906	20,808	0	7,272
3	0	21,816	0	161,208	9,396	14,544
4	0	7,272	14,544	0	120,906	20,808
5	9,396	7,272	0	29,088	0	161,208

$(\#_{P3} + \#_{P3M2} \cdot 1013) \bmod N$
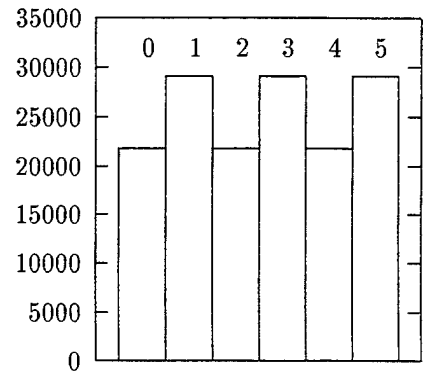
$\chi = 265,140$ (23.9% of η)
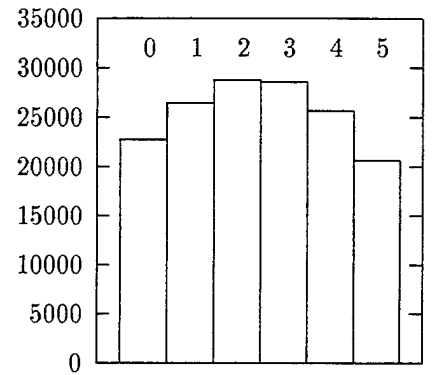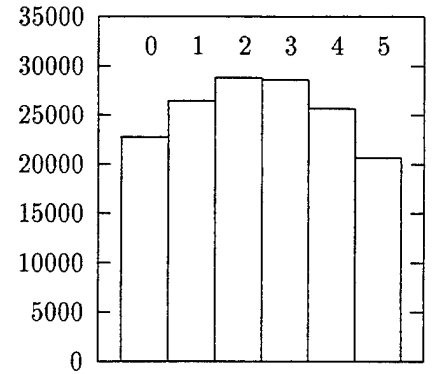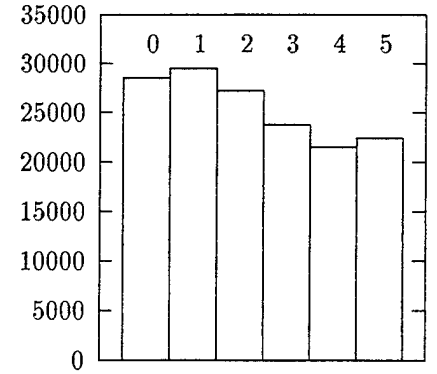
Figure 8: Number of arcs from process i to process j, $\eta^{i,j}$ ($N = 6$ and $k = 5$).

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Contractor Report

4. TITLE AND SUBTITLE
DISTRIBUTED STATE-SPACE GENERATION OF DISCRETE-STATE STOCHASTIC MODELS

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Gianfranco Ciardo
Joshua Gluckman
David Nicol

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 95-75

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-198233
ICASE Report No. 95-75

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report
Submitted to ORSA Journal on Computing

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
High-level formalisms such as stochastic Petri nets can be used to model complex systems. Analysis of logical and numerical properties of these models of ten requires the generation and storage of the entire underlying state space. This imposes practical limitations on the types of systems which can be modeled. Because of the vast amount of memory consumed, we investigate distributed algorithms for the generation of state space graphs. The distributed construction allows us to take advantage of the combined memory readily available on a network of workstations. The key technical problem is to find effective methods for on-the-fly partitioning, so that the state space is evenly distributed among processors. In this paper we report on the implementation of a distributed state-space generator that may be linked to a number of existing system modeling tools. We discuss partitioning strategies in the context of Petri net models, and report on performance observed on a network of workstations, as well as on a distributed memory multi-computer.

14. SUBJECT TERMS
Markovian Analysis; state-space generation; reliability analysis; petri nets

15. NUMBER OF PAGES
25

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		