# An Abstract Language for Specifying Markov Reliability Models

**Ricky W. Butler**
NASA Langley Research Center

*Key Words*—Markov model, Fault-tolerant system, Reliability modeling, Model specification

*Reader Aids*—
Purpose: Advance state of the art
Special math needed for explanation: Markov concepts
Special math needed to use results: None
Results useful to: Reliability theoreticians and analysts

*Abstract*—In principle, Markov models can be used to describe the reliability of virtually any fault-tolerant system. However, the process of delineating all of the states and transitions in a model of a complex system can be devastatingly tedious and error-prone. This paper presents a new approach to this problem by using an abstract model-definition language. The language essentially defines a set of rules which are used to generate the Markov model automatically. These rules correspond to the basic concepts used to create models of fault-tolerant systems. A small number of statements in the language can be used to describe a very large model. A variation in the system (such as in the number of initial spares) can be accomplished by changing only one line in the model definition, although such a change represents a large increase in the size of the Markov model. This high-level language is described in a non-formal manner and illustrated by several examples.

A computer program has been developed which translates the abstract language described in this paper into the input language for the SURE (Semi-Markov Unreliability Range Evaluator) program. The program has been named ASSIST (Abstract Semi-Markov Specification Interface to the SURE Tool). It is written in Pascal and runs on a VAX 11/750 in the NASA AIRLAB Facility.

## 1. INTRODUCTION

The reliability analysis of an aircraft or spacecraft electronic system is an essential part of both the design and the validation process. Traditional electronic systems were static, not relying on system reconfiguration for fault tolerance. For such systems, combinatorial mathematics were adequate to analyze system reliability. A graphic representation of such a combinatorial analysis, Fault-Tree Analysis, is frequently used by reliability engineers. Unfortunately, for reconfigurable systems, the fault-tree approach is inadequate. This is true whether the reconfiguration is by replacing a faulty unit with a spare or by removing the faulty unit and degrading to a lower level of redundancy. The more powerful Markov model must be used to analyze such systems. For more than a decade automated tools based on Markov methods have been developed [1, 2].

A new mathematical technique was developed enabling the efficient computation of such models [3]. This technique was embedded in the new reliability analysis tool called SURE [4]. The computational power of the tool enables it to process extremely complex models. However, the process of defining such models is still quite tedious. For well-structured systems, such as the Software Implemented Fault Tolerance (SIFT) system [5], very small and simple models can capture the essential fault tolerance behavior. Highly modular systems consisting of many statistically independent subsystems can be analyzed by dealing with each subsystem in a separate model and computing the overall system reliability with simple combinatorics. However, for large highly integrated systems a single very large model might be necessary. This problem is not related to any particular reliability analysis tool, but arises fundamentally from the complex nature of the system. The process of defining such a model can be tedious and error-prone. This paper describes a new approach to defining such models using an abstract model-definition language.

## 2. NOMENCLATURE

ASSIST Abstract Semi-Markov Specification Interface to the SURE Tool — a translator of the language of this paper into the SURE input language.

SIFT Software Implemented Fault-Tolerance — an experimental fault-tolerant computer system.

SURE Semi-Markov Unreliability Range Evaluator program — a program that computes the probability of entering a death-state of a semi-Markov model.

Death state an absorbing state in a Markov or semi-Markov model.

Reconfiguration the process of logically or physically removing a faulty processor from a system.

Semi-Markov model A generalization of a continuous-time Markov process where the time spent between transitions is not necessarily exponentially distributed.

Spare extra processor in a system; the spare is used to replace faulty processors.

Triad a set of three processors which execute the exact same program and use 3-way voting to mask errors. Also referred to as TMR, triple modular redundancy.

## 3. THE MODEL-DEFINITION CONCEPT

Modeling a fault-tolerant system is not an exact science; it is still very much an art. Reliability analysis must study a fault-tolerant architecture and capture the essential aspects of its design which contribute to its fault tolerance. For example, suppose:

0018-9529/86/1200-0595$01.00©1986 IEEE

1. We have a SIFT-like system consisting initially of six statistically-independent processors which have a constant failure rate λ.
2. Each processor executes the exact same program on exactly the same inputs so that all non-faulty processors produce exactly the same output. The system votes the outputs prior to external use. Thus, as long as a majority of the processors are non-faulty, any erroneous values are masked.
3. The system removes the faulty processors at constant (hazard) rate δ via reconfiguration.

The Markov model in figure 1 characterizes this system.

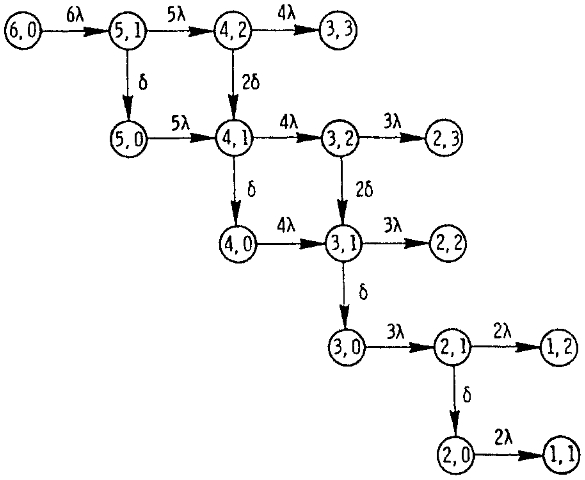

Fig. 1. Markov Model of SIFT-Like Architecture

The states of the model are defined by the ordered pair (NW, NF):

NW number of working processors currently in the configuration
NF number of faulty processors in the configuration.

The three assumptions are clearly manifested in the model. Assumption 1 (processors fail independently at rate λ) leads to horizontal transitions. Assumption 2 (system failure occurs when the number of failed processors is greater than or equal to the number of non-failed processors) is reflected in the death states of the model. Assumption 3 (reconfiguration occurs at rate δ) is represented in the vertical transitions.

It is the goal of the abstract model-definition language introduced in this paper to express concepts such as these so that an automatic generation of the corresponding Markov model is possible. Such a capability can be used in conjunction with a Markov model analysis program to provide a high-level reliability analysis work station. This concept is illustrated in figure 2.

A new program, Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST), has been developed at the Langley Research Center. This program translates the abstract model-definition language of this paper into the SURE input language. The SURE program input language is essentially a full enumeration of the translation matrix of the semi-Markov model.

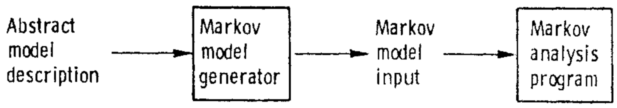

Fig. 2. Reliability Analysis Work-station Concept

## 4. THE ABSTRACT LANGUAGE

The language is not formally described. This paper does not specify the design of a translator but rather treats the problem of Markov model-description and a possible approach to the problem. Nevertheless, it is necessary to define a few conventions to facilitate the description of the language:

1. All reserved words are in italics.
2. Lowercase words which are surrounded by quotes, such as "const", indicate items which are to be replaced by something defined elsewhere.
3. Items enclosed in braces { } can be omitted or repeated as many times as desired.

The language consists of 5 types of statements each of which is discussed in the following sections.

1. The constant-definition statement
2. The *SPACE* statement
3. The *START* statement
4. The *DEATHIF* statement
5. The *TRANTO* statement

### 4.1 *The constant-definition statement*

A constant-definition statement equates an identifier consisting of letters and digits to a number. For example:

LAMBDA = 0.0052;

RECOVER = 0.005;

Once defined, an identifier can be used instead of the number it represents. In the following sections, the phrase "const" is used to represent a constant which can be either a number or a constant identifier. Constants can also be defined in terms of previously defined constants:

LAMBDA = 1*E* − 4;

GAMMA = 10 * LAMBDA:

In general the syntax is:

"ident" = "expression";

"ident" is a string of up to 8 characters and digits beginning with a character and "expression" is an arbitrary mathematical expression using constants and any of the following operations:

| | |
|---|---|
| + | addition |
| − | subtraction |
| * | multiplication |
| / | division |
| ** | exponentiation |
| = | equals |
| > | greater than |
| ⩾ | greater than or equal |
| < | less than |
| ⩽ | less than or equal |
| AND | logical and |
| OR | logical or |
| NOT | logical not |

and functions:

| | |
|---|---|
| EXP(X) | exponential function |
| LN(X) | natural logarithm |
| SIN(X) | sine function |
| COS(X) | cosine function |
| ARCSIN(X) | arcsine function |
| ARCCOS(X) | arccosine function |
| ARCTAN(X) | arctangent function |
| SQRT(X) | square root |

Both ( ) and [ ] can be used for grouping in the expressions. The following commands contain legal expressions:

ALPHA = 1E − 4;

RECV = 1.2 * EXP(− 3 * ALPHA);

DELTA = 1.2 * [(ALPHA + 2.3E − 5) * RECV
    + 1/ALPHA];

### 4.2 *The SPACE statement*

This statement is used to specify the state space on which the Markov model is defined. Essentially, the state space is defined by an $n$-dimensional vector where each component of the vector defines an attribute of the system being modeled. In the SIFT-like architecture example the state space is (NW, NF). This would be defined in the abstract language as

*SPACE* = (NW: 0..6, NF: 0..6);

The 0..6 represents the range of values over which the components can vary. The number of components (ie, the dimension of the vector space) can be as large as desired. In general the syntax is:

*SPACE* = ("ident": "const" ... "const"{, "ident":
    "const" ... "const"});

The identifiers, "ident", used in the SPACE statement are referred to as the: state space variables.

### 4.2 *The START statement*

This statement indicates which state is the start state of the model. This state corresponds to the initial state of the system being modeled, ie, the probability the system is in this state at time 0 is 1. In the SIFT-like architecture example the initial state is (6, 0). This is specified in the abstract language by:

*START* = (6, 0);

In general the syntax is:

*START* = ("const"{, "const"});

The dimension of the vector must be the same as in the *SPACE* statement.

### 4.4 *The DEATHIF statement*

The *DEATHIF* statement specifies which states are death states, ie, absorbing states in the model. The following is an example in the space (DIM1: 2..4, DIM2: 3..5):

*DEATHIF*(DIM1 = 4) OR (DIM2 =3);

This statement defines (4, 3), (4, 4), (4, 5), (2, 3), and (3, 3) as death states. In general the syntax is:

*DEATHIF*"expression";

The expression in this statement must be Boolean.

### 4.5 *The TRANTO statement*

This is the most important statement in the language. It is used to describe and consequently generate the model recursively. The following statement generates all of the fault-arrival transitions in the model of figure 1:

*IF* NW > 0 *TRANTO* (NW − 1, NF + 1)
    *BY NW* * LAMBDA;

The general syntax is:

*IF* "expression" *TRANTO* ("expression",
    {, "expression"}) *BY* "expression"

In all of the expressions of this statement the state space variables can be used. The value of a state space variable is the corresponding value in the source state to which the TRANTO statement is being applied. For example, if the TRANTO statement is being applied to state (4, 5) and the state space was defined by *SPACE* = (A: 0..10, Z: 2..15) then A = 4 and Z = 5. The first expression following the IF must be Boolean. Conceptually, it determines whether this rule applies to a particular state. For example, in the state space *SPACE* = (A1: 1..5, A2: 0..1), the expression (A1 > 3) AND (A2 = 0) is true for states (4, 0) and (5, 0) only. The vector following the *TRANTO* reserved word defines the destination state of the transition to be added to the model. Each expression within the parentheses must evaluate to an integer. For example, if the state space is (X1, X2) and the source state is (5, 3), then the vector (X1 + 1, X2 − 1) refers to (6, 2). The expression following the *BY* indicates the rate of the transition to be added to the model. This expression must evaluate to a real number.

The *TRANTO* statement is applied to every state in the model as described by the —

*Model-Generation Algorithm*

```
Initialize READY-SET to contain the start state only
WHILE READY-SET is not empty DO
  Select and remove a state from READY-SET
  IF the selected state does not satisfy a DEATHIF statement THEN
    Apply each TRANTO rule to the selected state as follows:
      IF the TRANTO if-expression evaluates to TRUE THEN
        Add the transition to the model.
        If the destination state is new, add it to the READY-SET
      ENDIF
  ENDIF
ENDWHILE
```

## 4.6 *Additional feature*

Comments are included in the language to increase the readability of a model description. Comments are placed between (* and *) and can be used anywhere that a blank character can be used.

## 5. EXAMPLE 1: SIFT-LIKE ARCHITECTURE

Now we can specify the model of figure 1 in the language:

*Model*

```
NP = 6; (* number of processors initially *)
LAMBDA = 1E - 4; (* fault arrival rate *)
DELTA = 3.6E3; (* recovery rate *)
SPACE = (NW: 0..NP, (* number working processors *)
         NF: 0..NP); (* number faulty processors *)
START = (NP, 0);
IF NW > 0 TRANTO (NW - 1, NF + 1) BY NW * LAMBDA; (* fault arrivals *)
IF NF > 0 TRANTO (NW, NF - 1) BY NF * DELTA; (* system recovery *)
DEATHIF NF ≥ NW; (* death if majority not working *)
```

The two *TRANTO* statements correspond to the first and third concepts used to define the model:

1. Every processor in the current configuration fails at rate $\lambda$.
3. The system removes faulty processors at rate $\delta$.

The *DEATHIF* statement corresponds to the second concept:

2. A majority of processors in the configuration must not have failed in order for the system to be safe.

The flexibility and power of this language can be seen by observing that only the NP = 6 statement would have to be changed in order to model a similar system which initially contains 9 processors.

## 6. EXAMPLE 2 — TRIAD WITH SPARES

A system consists of a triad of processors and a set of cold spares and the following 4 rules.

1. The active triad of processors performs 3-way voting so that the failure of a single active processor is masked and thus does not cause system failure.
2. The active processors fail at rate LAMBDA and the cold spares at a different rate GAMMA.
3. The system can detect a faulty active processor and is capable of two different types of reconfiguration. If a spare is available, the system is reconfigured by replacing the failed processor with a spare at (hazard) rate DELTA. If no spares are available, reconfiguration is accomplished by degrading to a simplex at (hazard) rate DEGRATE.
4. If a spare fails it remains undetected, so the reconfiguration process can bring in a faulty processor.

*Model*

```
NSI = 2; (* number of spares initially *)
LAMBDA = 5E - 4; (* failure rate of active processors *)
GAMMA = 2E - 5; (* failure rate of spares *)
DELTA = 3.6E3; (* rate faulty processors are removed *)
DEGRATE = 1.2E3; (* rate system degrades to a simplex *)
SPACE = (WP: 0..3, (* number of working processors *)
         FP: 0..3, (* number of failed active processors *)
         NS: 0..NSI, (* number of spares *)
         FS: 0..NSI); (* number of failed spares *)
START = (3, 0, NSI, 0);
IF WP > 0 (* a processor can fail *)
  TRANTO (WP - 1, FP + 1, NS, FS) BY WP * LAMBDA;
IF (NS - FS) > 0 (* a spare can fail *)
  TRANTO (WP, FP, NS, FS + 1) BY (NS - FS) * GAMMA;
IF (FP > 0) AND (NS > FS) (* a non-failed spare becomes active *)
  TRANTO (WP + 1, FP - 1, NS - 1, FS) BY (1 - FS/NS) * DELTA;
IF (FP > 0) AND (FS > 0) (* a failed spare becomes active *)
  TRANTO (WP, FP, NS - 1, FS - 1) BY FS/NS * DELTA;
IF (FP > 0) AND (NS = 0) (* no more spares, degrade to simplex *)
  TRANTO (1, 0, 0, 0) BY DEGRATE;
DEATHIF FP ≥ WP;
```

The transitions generated by this specification are:

```
(3, 0, 2, 0) → (2, 1, 2, 0) BY 3 * LAMBDA
(3, 0, 2, 0) → (3, 0, 2, 1) BY 2 * GAMMA
(2, 1, 2, 0) → (1, 2, 2, 0) BY 2 * LAMBDA
(2, 1, 2, 0) → (2, 1, 2, 1) BY 2 * GAMMA
(2, 1, 2, 0) → (3, 0, 1, 0) BY DELTA
(3, 0, 2, 1) → (2, 1, 2, 1) BY 3 * LAMBDA
(3, 0, 2, 1) → (3, 0, 2, 2) BY 1 * GAMMA
(2, 1, 2, 1) → (1, 2, 2, 1) BY 2 * LAMBDA
(2, 1, 2, 1) → (2, 1, 2, 2) BY 1 * GAMMA
(2, 1, 2, 1) → (3, 0, 1, 1) BY 0.5 * DELTA
(2, 1, 2, 1) → (2, 1, 1, 0) BY 0.5 * DELTA
(3, 0, 1, 0) → (2, 1, 1, 0) BY 3 * LAMBDA
(3, 0, 1, 0) → (3, 0, 1, 1) BY 1 * GAMMA
(3, 0, 2, 2) → (2, 1, 2, 2) BY 3 * LAMBDA
(2, 1, 2, 2) → (1, 2, 2, 2) BY 2 * LAMBDA
(2, 1, 2, 2) → (2, 1, 1, 1) BY DELTA
(3, 0, 1, 1) → (2, 1, 1, 1) BY 3 * LAMBDA
(2, 1, 1, 0) → (1, 2, 1, 0) BY 2 * LAMBDA
(2, 1, 1, 0) → (3, 0, 0, 0) BY DELTA
(2, 1, 1, 0) → (2, 1, 1, 1) BY GAMMA
(2, 1, 1, 1) → (1, 2, 1, 1) BY 2 * LAMBDA
(2, 1, 1, 1) → (2, 1, 0, 0) BY DELTA
(3, 0, 0, 0) → (2, 1, 0, 0) BY 3 * LAMBDA
(2, 1, 0, 0) → (1, 2, 0, 0) BY 2 * LAMBDA
(2, 1, 0, 0) → (1, 0, 0, 0) BY DEGRATE
(1, 0, 0, 0) → (0, 1, 0, 0) BY 1 * LAMBDA
```

By changing the constant NSI, the effect of using more spares can be investigated. The size and complexity of the state space grows appreciably as this constant is increased, yet is easily accomplished via the abstract language:

| NSI | # states | # transitions |
|---|---|---|
| 2 | 20 | 26 |
| 3 | 32 | 46 |
| 5 | 65 | 104 |
| 10 | 200 | 354 |

## 7. EXAMPLE 3 — TWO TRIADS WITH SPARES

The system characteristics are the same as example 2 except for the additional assumptions 5-7.

5. Both triads are necessary for successful system operation. Therefore, if two active processors are faulty in either triad the system fails.

6. As long as spares are available, a faulty processor in a triad is replaced from the spares pool. If no spares are available, then a triad is collapsed into a simplex and the other good processor is added to the spares pool.

7. Spares fail at a different rate than active processors. For simplicity, the failed spares are self announcing, viz, immediately recognized as failed (say by an off-line diagnostic) and thus are not brought into the active configuration. (Models of systems using imperfect spare diagnostics can be modeled using this language but would make this example more complex).

*Model*

```
NSI = 3; (* Number of spares initially, can be anything *)
SPACE = (N1: 0..3, (* Number of processors in first triad *)
         N2: 0..3, (* Number of processors in second triad *)
         F1: 0..2, (* Number of faulty processors in first triad *)
         F2: 0..2, (* Number of faulty processors in second triad *)
         NS: 0..NSI); (* Number of spares *)
START = (3, 3, 0, 0, NSI);
LAMBDA = 5E - 4; (* failure rate of active processors *)
GAMMA = 2E - 5; (* failure rate of spares *)
DELTA = 3.6E3; (* rate faulty processors are replaced w/spares *)
DEGRATE = 1.2E3; (* rate at which a triad is degraded to a simplex *)
DEATHIF (2 * F1 > N1) OR (2 * F2 > N2);
IF N1 > 0 TRANTO (N1, N2, F1 + 1, F2, NS) BY (N1 - F1) * LAMBDA;
IF N2 > 0 TRANTO (N1, N2, F1, F2 + 1, NS) BY (N2 - F2) * LAMBDA;
IF NS > 0 TRANTO (N1, N2, F1, F2, NS - 1) BY NS * GAMMA;
IF (F1 > 0) AND (NS > 0)
   TRANTO (N1, N2, F1 - 1, F2, NS - 1) BY DELTA;
IF (F2 > 0) AND (NS > 0)
   TRANTO (N1, N2, F1, F2 - 1, NS - 1) BY DELTA;
IF (F1 > 0) AND (NS = 0) TRANTO (1, N2, 0, F2, NS + 1) BY DEGRATE;
IF (F2 > 0) AND (NS = 0) TRANTO (N1, 1, F1, 0, NS + 1) BY DEGRATE;
```

The *DEATHIF* statement specifies that the system fails if a majority of processors fail in either triad. The first three *TRANTO* statements specify fault arrivals in the two triads and spares. The next *TRANTO* statement specifies recovery when spares are available. The last *TRANTO* statement describes the recovery process when no spares are available. This specification generates a 58-state model with 89 transitions.

## APPENDIX

### Language Extensions

The fundamental concept for an abstract specification language for Markov models has been developed in the main body of this paper. The constructs of the language have adequate expressive power to describe complex systems with a minimal number of statements. However, there are many possible extensions to this language which can further simplify the model-description process.

*Extension 1: Array State Variables.*

The basic language allows the definition of state space variables with a *SPACE* statement. The language can easily be extended to allow an array of state space variables as follows:

```
SPACE = (NW: ARRAY[1..3] OF 0..6,

         NF: ARRAY[1..3] OF 0..3);
```

This statement creates a 6-dimensional space. The state space variables are NW[1], NW[2], NW[3], NF[1], NF[2], NF[3].

*Extension 2: FOR Statement.*

Many times several *TRANTO* statements are needed which are identical except they operate on different state space variables. The FOR statement defines several *TRANTO* rules at once:

```
SPACE = (NW: ARRAY[1..5] OF 0..6,

         NF: ARRAY[1..5] OF 0..3);

FOR I = 1, 5

  IF NW[I] > 0 TRANTO NF[I] = NF[I]

      + 1 BY LAMBDA;

ENDFOR;
```

This *FOR* statement is equivalent to five *TRANTO* statements, one for each value of I in the range from 1 to 5. The assignment statement after the *TRANTO* reserved word replaces the vector of the basic *TRANTO* statement. This statement defines the destination state of each new transition by specifying the change in a state space variable from the source to destination state. There can be as many of these assignment statements after the *TRANTO* reserved word and before the *BY* reserved word as there are variables in the state space.

*Extension 3: Nested IF THEN ELSE.*

The *IF* expression of the *TRANTO* statement can be extended in the obvious way:

```
IF "expression" THEN
  IF "expression" THEN
    TRANTO "vector" BY "expression";
    TRANTO "vector" BY "expression";
    TRANTO "vector" BY "expression";
  ENDIF
ELSE
    TRANTO "vector" BY "expression";
    TRANTO "vector" BY "expression";
ENDIF;
```

## ACKNOWLEDGMENT

I am grateful to Sally Johnson, (the programmer of the ASSIST program), for her review of this paper, her many helpful suggestions concerning the language, and the use of her prototype version of ASSIST in the generation of the example problems.

## REFERENCES

[1] Robert M. Geist, Kishor S. Trivedi, "Ultrahigh reliability prediction in fault-tolerant computer systems", *IEEE Trans. Computers,* vol C-32, 1983 Dec, pp 1118-1127.

[2] Srinivas V. Makam, Algirdas Avizienis, "ARIES 81: A reliability and life-cycle evaluation tool for fault-tolerant systems", *Proceedings of the Fault-Tolerant Computing Symposium*, vol 12, 1982.

[3] Allan L. White, "Upper and lower bounds for semi-Markov reliability models of reconfigurable systems", *NASA CR-172340,* 1984.

[4] Ricky W. Butler, "The semi-Markov unreliability range evaluator (SURE) program", *NASA TM-86261*, 1984 July.

[5] Jack Goldberg, et al, "Development and analysis of the software implemented fault tolerance (SIFT) computer", *NASA CR-172146,* 1984.

## AUTHOR

Ricky W. Butler; NASA Langley Research Center; MS 130; Hampton, Virginia 23665 USA.

**Ricky W. Butler** is a research engineer at the Langley Research Center. He received his BA degree from the University of Virginia in Mathematics in 1976 and his MS degree in Computer Science from the University of Virginia in 1978. His research interests are in the design and validation of fault-tolerant computer systems used for flight-critical applications.

Manuscript TR85-126 received 1985 November 29; revised 1986 August 25. ★★★

GOEL, ET AL.: COST ANALYSIS OF A 2-UNIT PRIORITY STANDBY SYSTEM

*(continued from page 585)*

3. When P-unit fails, the standby unit is switched to operate. The switch is perfect at the time of need with probability $p$.

4. A single repairman is available, to repair a failed unit or the switch, instantaneously at the time of need with probability $b$.

5. The distributions of failure time, repair time, and repairman availability time (if it is not available instantaneously) are general.

*Supplement*

Detailed derivations and results are given in a separately available Supplement:

NAPS document No. 04412-F; 21 pages in this Supplement. For current ordering information, see "Information for Readers & Authors" in a current issue. Order NAPS document No. 04412, 84 pages. ASIS-NAPS; Microfiche Publications; POBox 3513, Grand Central Station; New York, NY 10163 USA.

## AUTHORS

L. R. Goel, Head; Department of Statistics; Institute of Advanced Studies; Meerut University; Meerut - 250 005 INDIA.

Rakesh Gupta, Lecturer; Department of Statistics; Institute of Advanced Studies; Meerut University, Meerut - 250 005 INDIA.

S. K. Singh, Reader; Department of Mathematics and Statistics; Ravi Shanker University; Raipur - 492 010 INDIA.

Manuscript TR85-047 received 1985 June 6; revised 1986 June 2. ★★★