NASA Technical Memorandum 87735 NASA-TM-87735 19860020975

ASSIST User's Manual

Sally C. Johnson

August 1986

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

CONTENTS

INTRODUCTION

Semi-Markov models can be used to calculate the reliability of virtually any fault-tolerant system. New advances in computation, such as the Semi-Markov Range Evaluator (SURE) program, enable the accurate solution of extremely large and complex semi-Markov models (refs. 1 and 2). However, the generation by hand of the large models needed to capture the complex failure and reconfiguration behavior of most realistic fault-tolerant architectures has been an intractable problem. Much research has been done on techniques for model pruning and state aggregation to simplify the models, at the expense of accuracy (refs. 3 and 4).

Creating semi-Markov models that accurately capture the fault behavior of complex systems is tedious and error-prone. However, often even the most complex characteristics of a system can be described by relatively simple rules. The models only become complex because these few rules combine many times to form models with large numbers of states and transitions between them. The Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST) computer program allows the user to specify the behavior rules of the model in an abstract language, then the semi-Markov model is generated automatically from the rules. The abstract language used by ASSIST and the model generation method were developed by Ricky Butler (ref. 5). The semi-Markov model is output in the format required for input to the SURE program. For semi-Markov analysis programs requiring a different form of input for the semi-Markov model than SURE, a simple program could be written to modify the model description file.

The abstract input language for ASSIST is described in the following section. Analysis of a sample fault-tolerant architecture shows how the behavior of a system can be captured by a few general rules. The automatic model generation process is then explained. The syntax of the ASSIST input language is then described in detail, and several example problems are given.

SEMI-MARKOV MODELING

A semi-Markov model consists of a number of system states and transitions between them. Each state is defined by a state vector, where each element of the vector takes on an integer value within a defined range. An element can represent any meaningful characteristic, such as the number of good components of one type in the system, or the number of faulty components of another type in use. Each element is assigned an appropriate variable name for ease of reference. The state space variables for the model and their valid ranges are defined in the "space" statement. The user specifies the initial system state in the "start" statement. This establishes the initial values of the state space variables for the generation of the system model.

The "death" conditions of the model must be defined in terms of state space values. These "death" conditions could be system failure or the onset of degraded performance operation or other situations resulting from failures.

The transitions between states in the model are specified using transition expressions. These expressions have three main parts. The first part is a boolean expression to describe the state space variable values of states for which the transition is appropriate. The second part defines the destination state for the transition in terms of the state space variable values. The third part defines the rate at which the transition occurs.

The sample architecture consists of a triad of processors each executing the same program plus a pool of two cold spare processors. Each of the three processors receives identical inputs so all non-faulty processors produce the same output, and the three outputs are voted. Any incorrect outputs are masked by the voting as long as a majority of the active processors are non-faulty. A faulty processor is detected by the voter and is replaced with a cold spare processor if one is available. There is no fault detection for spare processors until they become active. The semi-Markov model to describe this system is shown in figure 1.

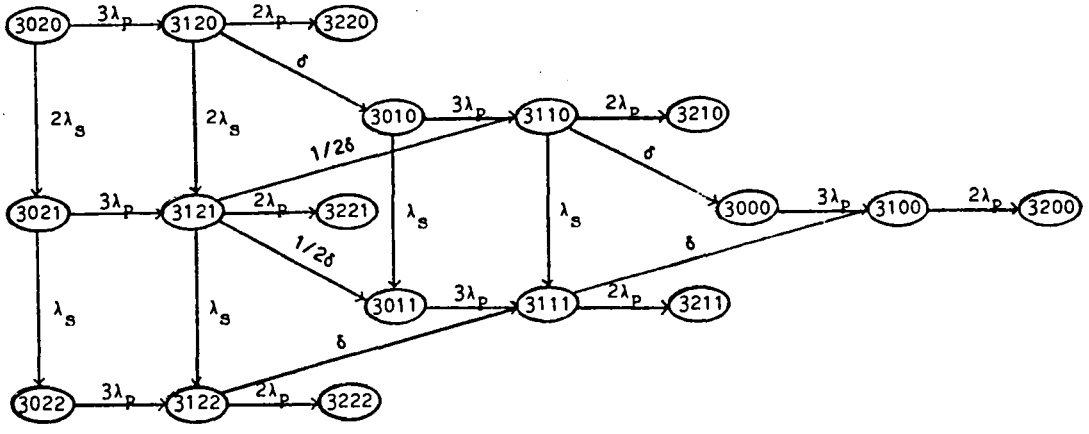

Figure 1. Semi-Markov model for a triad with two cold spares.

The states in the example model are described by the vector (NP, NFP, NS, NFS), where

NP = Number of active processors,
NFP = Number of failed active processors,
NS = Number of cold spare processors, and
NFS = Number of failed cold spare processors.

The fault and recovery behavior of the example system is described by the following rules:

1. The failure rates of each active processor is λ_p.
2. The failure rate of the cold spare processors is λ_s.
3. A failed active processor is replaced by a spare processor at rate δ.
4. System failure occurs unless a majority of the active processors are non-faulty.

Rules 1 through 3 above describe the transitions possible between states in the semi-Markov model. The fourth rule describes the "death" states of the model. The example system starts with 3 non-faulty, active processors and 2 non-faulty, cold spare processors, thus the start state is (3, 0, 2, 0).

ASSIST MODEL GENERATION

The ASSIST program builds the model from the initial "start" state by recursively applying the transition rules. Before application of a rule, ASSIST checks all of the "death" conditions to see if the current state is a "death" state. Since a "death" state denotes system failure, no transitions can leave a "death" state. All of the transition rules are then evaluated, and transitions to new states are generated where appropriate. When all possible branches terminate in a "death" state, model building is complete. The output file contains a definition of each transition and its rate. A listing file is also generated to assist the user in determining whether the model generated describes the intended system behavior.

The specific algorithm used to generate the model is as follows. The program maintains a READY SET of states to be processed. Initially, the READY SET contains only the start state of the model. Each state in the READY SET is processed in the following manner. If the state meets any of the "death" conditions, then that state is a "death" state, and no transitions can leave it, so the state is removed from the READY SET. If the state is not a "death" state, then each transition rule is applied to the state in the following manner to generate all possible transitions leaving the state. If the condition expression of the transition rule is true for the current state, then the destination state description in the rule is used to determine the destination state. If this state is within the bounds of the state space parameters, then this is a valid transition. If the destination state has not already been defined in the model, a unique integer is assigned to the state, and it is added to the READY SET. If the destination state was already

defined in the model, then it was placed in the READY SET for processing when it was first defined. The rate of the transition is determined from the rate expression, and the transition description is printed to the SURE model file. After all of the transition rules have been applied to the state, it is removed from the READY SET.

ASSIST FILES

The ASSIST program reads an input file containing the model definition rules and creates two output files: the model file and an optional listing file. The model file (.MOD) is the SURE input file. To make the model easier to understand, this file is annotated with the state space variables of each state in comments, which SURE will ignore. For example:

```
1(* 6,0,0 *),   2(* 5,1,0 *) = LAMBDA;
```

This feature can be turned off by adding the line `COMMENT=0;` to the ASSIST input file.

In addition to a listing of the ASSIST input file, the listing file contains a list of the destination state of each arc leaving each non-death state in the model. Destination states that are death states are annotated with an asterisk. The listing file also contains a list of the mappings between the SURE state numbers and the state variables of that state in ASSIST. The model file and listing file generated for the example problem stated above may be found in Appendix B.

The ASSIST program is executed by entering ASSIST followed by from 0 to 3 parameters, separated by commas, for example:

```
ASSIST FOO, MARY, JOE
```

This specifies the input file as FOO.AST, the model file as MARY.MOD, and the listing file as JOE.LIS. Extents for the file names may be specified instead of using the default extents. If the second parameter is missing, the first parameter is also used for the model file, but with the .MOD extent. If the third parameter is missing, the model file name is used for the listing file, but with the .LIS extent. For example,

```
ASSIST FOO.X
```

specifies the input file as FOO.X, the model file as FOO.MOD, and the listing file as FOO.LIS.

If the ASSIST command is entered with no parameters, then the user is queried for the names of the input, model, and listing files. The user is also given the option of no listing file.

The input file is then read in and the commands are parsed. If any errors are found during parsing, the line with the error is printed to the screen, and an error message is printed on the screen and in the listing file, and then parsing continues. Appendix A contains a list of the error messages printed by ASSIST with a description of each error. If no errors are found during the parsing phase, the semi-Markov model is automatically generated. After the model is generated, the model statistics are printed and the program terminates. The statistics printed are: the processing time (the execution time for generating the model after parsing of the input file was completed), the number of states in the model, and the number of transitions in the model.

ASSIST COMMAND SYNTAX

It is necessary to define a few conventions to facilitate the description of the language:

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

The language consists of the following types of statements:

Constant definition statement
INPUT statement
Variable definition statement
SPACE statement
START statement
DEATHIF statement
TRANTO statement
FOR statement
SURE statement
Comments

Each of these statements will be discussed in the following sections.

Constant Definition Statement

A constant definition statement equates an identifier name to a number. For example:

```
N_PROCS = 8;
LAMBDA = 0.0052;
```

If a state space variable is used in the definition, then the identifier is a variable identifier, as described in the next section, not a constant identifier. Once defined, a constant identifier may be used instead of the number it represents. In the following sections, the phrase "const" will be used to represent a constant which can be either a number or a constant identifier. Real constants may only be used in the rate specifications of

TRANTO statements. Constants may also be defined in terms of previously defined constants:

```
LAMBDA = 1E-4;
GAMMA = 10*LAMBDA;
```

In general the syntax is

```
"ident" = "expression";
```

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

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

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 [ ] may be used for grouping in the expressions. The following commands contain legal expressions:

```
ALPHA = 1E-4;
RECV = 1.2*EXP(-3*ALPHA);
DELTA = (ALPHA + 2.3E-5)*RECV;
```

Constants can also be one-dimensional arrays. The syntax for defining an array constant is:

```
"ident" = ( "const" {, "const" });
```

where each "const" defines the value of one member of the array. Repetition can also be used in defining an array constant, with the following syntax:

```
"const" OF "const"
```

where the first "const" defines the number of repetitions, and the second "const" is the constant value that is repeated. For example, the following statements define identical arrays:

```
FOO = (2 OF 3, 1, 3 OF 5);
FOO = (3, 3, 1, 5, 5, 5);
```

When values of array constants are referenced, the array index must be specified in square brackets, for example:

```
FOO[3]
```

refers to the third value in array FOO.

Certain user-defined constants have special meaning to the ASSIST program. These constants may be changed by constant definition or by using the INPUT command and inputting the new constant during ASSIST processing. These constants are:

ECHO. - As each line of the input file is processed, it is printed in the listing file with its line number. If the line "ECHO = 1;" is included in the input file, then subsequent input lines are also echoed to the screen during processing unless the line "ECHO = 0;" is processed.

COMMENT. - To make the model easier to understand, this file is annotated with the state space variables of each state in comments, which SURE will ignore. This feature can be turned off by adding the line COMMENT=0; to the ASSIST input file.

ONEDEATH. - If the statement "ONEDEATH = 1;" is included in the ASSIST input file, then in the SURE input file all of the death states will be numbered as state 0. The ASSIST .LIS file and the comments in the SURE input file will still refer to each death state by its unique state space variables. This feature is useful for decreasing the number of states in very large models. The default, "ONEDEATH = 0;", causes ASSIST to generate distinct death states.

INPUT Statement

This statement specifies that the user should be queried for the values of the one or more specified identifiers. Interactive input for identifiers RHO and DELTA may be specified as follows:

```
INPUT RHO,DELTA;
```

In general the syntax is:

```
INPUT "ident" {,ident} ;
```

When an input statement is processed, the user is queried interactively for the values. For the above example, the user is first prompted:

```
RHO?
```

The user should enter an integer or real value. The user is then prompted for the next identifier:

DELTA?

The user should again enter an integer or real value.

Variable Definition Statement

A variable definition statement equates an identifier name to an expression of constants and state space variables. For example:

```
SPACE = (NP: 0..4, NF: 0..4);
NX = 3*NF - 1;
WORKING = NP ⌐ NF;
```

Once defined, a variable identifier may be used instead of the expression it represents. In the following sections, variable identifiers may be used in statements wherever "expression"'s are called for, but not where "const"'s are needed. Real variables may only be used in the rate specifications of TRANTO statements. Variables may also be defined in terms of previously defined variables. In general the syntax is

```
"ident" = "expression";
```

where "ident" is a string of up to 8 characters, digits, and underscores (_) beginning with a character and "expression" is an arbitrary mathematical expression using constants and state space variables and any of the operations valid for constant identifiers.

If an array state space variable is used in the definition, an array index specifying a specific member of the array must be used. Array variable definitions are not allowed. For example:

```
SPACE = (NP: ARRAY[1..3]);
TOTAL_NP = NP[1] + NP[2] + NP[3];  (* is valid, and *)
XNP = NP[1]*3 -1;                  (* is valid, but *)
YNP = NP*3 - 1;                    (* is not allowed *)
```

The third definition above is not allowed because YNP is defined as a function of the entire array NP, which implicitly defines YNP to be an array.

SPACE Statement

This statement is used to specify the state space on which the semi-Markov model is defined. The state space is defined by a n-dimensional vector where each component of the vector defines an attribute of the system being modeled. In the example problem above the state space is (NP,NFP,NS,NFS). This would be defined in ASSIST by the statement

```
SPACE = (NP: 0..3, NFP: 0..3, NS: 0..2, NFS: 0..2);
```

The 0..3 represents the range of values over which the variable NP can vary. The number of components (i.e., 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 will be referred to as the "state space variables". The phrase :"const" .. "const" after each variable name is optional. Each variable with an unspecified range has the default range from -32768 to 32767, which is the range of integers on the VAX.

State space variables can also be arrays. The array is specified in the space statement as follows:

"ident" : ARRAY ["const" .. "const"] OF "const" .. "const"

The range phrase OF "const" .. "const" is optional with the same default range as above. For example:

```
N_PROCS = 3;
SPACE = (NC: ARRAY [1..N_PROCS] OF 0..6, NF: ARRAY [1..2], NX);
```

This statement creates a 6-dimensional space. The state space variables are NC[1], NC[2], and NC[3] with values ranging from 0 through 6, and NF[1] and NF[2] and NX with default ranges.

The current values of the state space variables may be used in expressions in the DEATHIF and TRANTO statements during model generation. For array variables, the array index must be specified in square brackets, for example:

```
NC[I]
```

START Statement

This statement indicates which state represents the start state of the model. This state corresponds to the initial state of the system being modeled, i.e. the probability the system is in this state at time 0 is 1. In the example architecture described above the initial state is (3, 0, 2, 0). This is specified in the abstract language by the following:

```
START = (3, 0, 2, 0);
```

In general the syntax is:

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

The dimension of the vector must be the same as the state space dimension defined in the SPACE statement.

To make variable-sized arrays more usable, repetition may be used in the START statement, as in the following example:

```
INPUT NX;
SPACE = (NP: ARRAY[1..2], NF, NC: ARRAY[1..NX] OF 1..6);
START = (2 OF 6, 0, NX OF 0);
```

This START statement fills array NP with 6's, sets NF to 0, and fills array NC with 0's.

DEATHIF Statement

The DEATHIF statement specifies which states are death states, i.e. 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 "boolean expression".
```

where a "boolean expression" is any mathematical expression that evaluates to TRUE or FALSE. A boolean expression may contain any of the operations valid for an "expression" plus any of the following operations:

```
=    equals
>    greater than
>=   greater than or equal
<    less than
<=   less than or equal

AND  logical and
OR   logical or
NOT  logical not
```

DEATHIF statements may be included inside FOR loops, as explained in the section "FOR Statements", but they may not be included within TRANTO statements as described in the next section.

TRANTO Statement

The TRANTO statement is the most important statement in the language. It is used to describe the state transitions in the model. The model is generated by applying the TRANTO rules to each state in the model in a recursive manner. The TRANTO statement consists of three basic parts: the condition expression, the destination state, and the rate expression.

The possible syntaxes for TRANTO statements are:

```
IF "cond expression" TRANTO "dest state" BY "rate expression";
```

or the condition expression of the TRANTO statement can be nested as follows:

```
IF "cond expression" THEN
   { "multiple TRANTO statements or TRANTO clauses" }
ELSE
   { "multiple TRANTO statements or TRANTO clauses" }
ENDIF;
```

or without the optional ELSE clause:

```
IF "cond expression" THEN
        { "multiple TRANTO statements or TRANTO clauses" }
ENDIF;
```

where TRANTO clauses are of the form:

```
TRANTO "dest state" BY "rate expression";
```

A TRANTO clause may not appear by itself without a condition expression. If the IF is not followed by a THEN, then only one TRANTO clause may be included, and no ELSE clause or ENDIF may be used. If the IF is followed by a THEN, then an optional ELSE clause may be included, and the IF statement must be terminated with an ENDIF. The THEN clause and the optional ELSE clause may contain multiple TRANTO clauses and/or nested TRANTO statements. Every rate expression must be followed by a semicolon, and the end of the entire nested statement must be followed by a semicolon.

For example:

```
IF "cond expression" THEN
    IF "cond expression" THEN
       TRANTO "dest state" BY  "rate expression";
       TRANTO "dest state" BY  "rate expression";
       TRANTO "dest state" BY  "rate expression";
    ENDIF
ELSE
    TRANTO "dest state" BY  "rate expression";
    TRANTO "dest state" BY  "rate expression";
ENDIF;
```

In all of the expressions of this statement the state space variables may 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 currently 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 Condition Expression. - The first expression following the IF is the condition expression. Condition expressions must be boolean expressions, as described in the section "DEATHIF Statement". Conceptually, the condition expression determines which states of the model this rule will be applied to. 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. Each condition expression followed by a THEN applies to every TRANTO clause until the matching ELSE or ENDIF. The negation of the condition expression applies to all TRANTO clauses between the ELSE and the ENDIF. Each condition expression not followed by a THEN applies only to the first TRANTO clause.

The Destination State. - The vector following the TRANTO reserved word defines the destination state of the transition to be added to the model. The destination state of a TRANTO statement can be specified using positional or assigned values.

Specification by positional values is as follows:

```
( "const" {, "const"})
```

where the dimension of the vector must be the same as the state space dimension defined in the SPACE statement. Each constant 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).

Specification by assigned values is as follows:

```
"ident" = "const" {, "ident" = "const"}
```

The assignments define the destination state of the transition by specifying the change in one or more state space variables from the source to destination state. There can be as many of these assignments as there are state space variables.

The two types cannot be mixed in the same statement. When assigned values are used, the parentheses are not used and state space variables which do not change values need not be specified. For example, the two TRANTO statements below are equivalent:

```
SPACE = (NP: 0..6, NF: ARRAY[1..3] OF 0..6, NX);
IF NF[2]>0 TRANTO (NP-1, NF[1], NF[2]-1, NF[3], NX) BY LAMBDA;
IF NF[2]>0 TRANTO NP = NP-1, NF[2] = NF[2]-1 BY LAMBDA;
```

The Rate Expression. - The expression following the BY indicates the rate of the transition to be added to the model. This expression may contain FOR variables and state space variables. The rate expression is the only expression in which real constants may be used. There are 3 ways of expressing rate in the SURE input language that are supported in ASSIST.

Slow transitions are specified by the transition rate. The syntax is:

```
"const"
```

where "const" is a real constant. Fast transitions may be specified by two different methods: White's method or the fast exponential method.

The syntax for White's method is:

```
<"mu", "sig", "frac">
```

where

"mu" = an expression defining the conditional mean transition time,
"sig" = an expression defining the conditional standard deviation of the transition time, and
"frac" = an expression defining the transition probability.

These three parameters are all real constants. The third parameter is optional. If omitted, the transition probability is assumed to be 1.0.

The syntax of the fast exponential rate expression is:

FAST "rate"

where "rate" is a real constant expression. The SURE program automatically calculates the conditional moments from the unconditional rates given in this expression. In the simple case with only one transition leaving a state, the following two rate expressions are equivalent:

$\langle 1/\alpha, 1/\alpha, 1 \rangle$

and

FAST α.

For more information on specifying the transition rates, see reference 2.

FOR Statement

Many times several TRANTO or DEATHIF statements are needed which are identical except they operate on different state space variables. The FOR statement defines several TRANTO or DEATHIF rules at once. The syntax is as follows:

```
FOR "ident" = "const" , "const"
      {multiple TRANTO and DEATHIF statements may go here}
ENDFOR;
```

The loop variable "ident" may only be referenced by statements within the FOR loop. The TRANTO and DEATHIF statements between the FOR and ENDFOR statements are processed with the loop variable "ident" varying between the range of integers specified. FOR statements may be nested, as in the following example:

```
SPACE = ( NC: ARRAY[1..5] OF 0..6,
          NF: ARRAY[1..5] OF 0..3 );

FOR I = 1,5
   FOR J = 1,2
      IF NC[I] > J TRANTO NF[I] = NF[I]+1 BY J*LAMBDA;
   ENDFOR;
   DEATHIF NC[I] < NF[I];
ENDFOR;
```

Each ENDFOR statement matches with the most recently preceding FOR statement. These FOR statements generate 10 TRANTO rules, one for each pair of (I,J) values (1,1), (1,2), (2,1), ... , (5,2). Five DEATHIF conditions would be defined using values of I from 1 to 5.

An IF-THEN-ELSE statement may be nested within a FOR loop, but a FOR loop cannot be nested within an IF-THEN-ELSE statement.

SURE Statement

Statements in the ASSIST input file that are put inside quotes are copied into the SURE input file and are not otherwise processed by ASSIST. For example:

```
"INPUT DELTA;"
```

or `"FOO = 1 TO 10 BY 2;"`

or
```
"(* THIS IS A LONG COMMENT TO BE *)
 (* INCLUDED IN THE SURE INPUT FILE *)"
```

The statements in quotes need not be followed by a semicolon for ASSIST. However, for the statement to be followed by a semicolon in the SURE input file, a semicolon must be put inside the quotes in the ASSIST input file. These statements are put in the front of the SURE input file with the constant definitions before the transition descriptions.

Comments

Comments in ASSIST must be initiated with "(*" and terminated with "*)". A comment may be anywhere in the input, even in the middle of an ASSIST statement, and may run as long as desired -- even on multiple lines.

EXAMPLES

In this section the use of ASSIST to generate semi-Markov models will be illustrated by several examples.

Example 1. Triad With Cold Spares

The example architecture described in the introduction may be described using ASSIST as follows:

```
N_PROCS = 3;                          (* Number of active processors *)
N_SPARES = 2;                         (* Number of cold spare processors *)
LAMBDA_P = 1E-4;                      (* Failure rate of active processors *)
LAMBDA_S = 1E-5;                      (* Failure rate of spare processors *)
DELTA = 3.6E3;                        (* Reconfiguration rate *)

SPACE = (NP:  0..N_PROCS,             (* Number of active processors *)
         NFP: 0..N_PROCS,             (* Number of failed active processors *)
         NS:  0..N_SPARES,            (* Number of spare processors *)
         NFS: 0..N_SPARES);           (* Number of failed spare processors *)

START = (N_PROCS, 0, N_SPARES, 0);

DEATHIF 2 * NFP >= NP;

IF NP > NFP TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_P;
   (* Active processor failure *)

IF NS > NFS TRANTO NFS = NFS+1 BY NS*LAMBDA_S;
   (* Spare processor failure *)

IF (NFP > 0 AND NS > 0) THEN
   IF NS > NFS TRANTO (NP, NFP-1, NS-1, NFS) BY (1-(NFS/NS))*NFP*DELTA;
      (* Replace failed processor with working spare *)

   IF NFS > 0 TRANTO (NP, NFP, NS-1, NFS-1) BY (NFS/NS)*NFP*DELTA;
      (* Replace failed processor with failed spare *)
ENDIF;
```

The TRANTO statements describe the three types of transitions possible between states in the semi-Markov model:

1. The failure rates of each active processor is λ_p.
2. The failure rate of the cold spare processor is λ_s.
3. A failed active processor is replaced by a spare processor at rate δ.

The third type of transition requires a more complicated TRANTO statement because the spare processor may or may not have failed before reconfiguration.

The DEATHIF statement describes the "death" condition for the model:

4. System failure occurs unless a majority of the active processors are non-faulty.

By changing the value of N_SPARES, a similar system with a different number of initial spares may be modeled.

The listing file and model file generated by ASSIST for this example are shown in Appendix B.

Example 2. Two Triads With a Pool of Spares

The example above can be expanded to model a system with several triads and a pool of spares using array state space variables. If two or more processors in an active triad fail then the system fails. As long as spares are available, a faulty processor in a triad is replaced from the spare pool. If no spares are available, then the triad is broken up and the nonfaulty processors are added to the spare pool.

This example is very similar to the first example, except that the DEATHIF statement and TRANTO statements pertaining to triads must be put inside FOR loops so that all triads are handled. The only significant changes to the model are a new transition type and a new type of system failure. The new transition is the breakup of a triad when it fails and there are no spares. System failure by exhaustion must also be modeled, which requires an extra state space variable and a new DEATHIF statement.

```
INPUT N_TRIADS;              (* Number of triads initially *)
INPUT N_SPARES;              (* Number of cold spare processors *)
N_PROCS = 3;                 (* Number of active processors *)
LAMBDA_P = 1E-4;             (* Failure rate of active processors *)
LAMBDA_S = 1E-5;             (* Failure rate of spare processors *)
DELTA1 = 3.6E3;              (* Reconfiguration rate to switch in spare *)
DELTA2 = 5.1E3;              (* Reconfiguration rate to break up a triad *)

SPACE = (NP:  ARRAY[1..N_TRIADS] OF 0..N_PROCS,
                             (* Number of active processors per triad *)
         NFP: ARRAY[1..N_TRIADS] OF 0..N_PROCS,
                             (* Number of failed active processors per triad*)
         NS,                 (* Number of spare processors *)
         NFS,                (* Number of failed spare processors *)
         NT:  0..N_TRIADS); (* Number of non-failed triads *)

START = (N_TRIADS OF N_PROCS, N_TRIADS OF 0, N_SPARES, 0, N_TRIADS);
```

```
IF NS > NFS TRANTO NFS = NFS+1 BY NS*LAMBDA_S;
   (* Spare processor failure *)

FOR J=1,N_TRIADS
   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1 BY (NP[J]-NFP[J])*LAMBDA_P;
      (* Active processor failure *)

   IF NFP[J] > 0 THEN
      IF NS > 0 THEN
         IF NS > NFS TRANTO NFP[J] = NFP[J]-1, NS = NS-1
            BY (1-(NFS/NS))*NFP[J]*DELTA1;
            (* Replace failed processor with working spare *)

         IF NFS > 0 TRANTO NS = NS-1, NFS = NFS-1 BY (NFS/NS)*NFP[J]*DELTA1;
            (* Replace failed processor with failed spare *)

      ELSE
         TRANTO NP[J] = 0, NFP[J] = 0, NS = NP[J]-NFP[J], NT = NT-1 BY DELTA2;
            (* Break up a failed triad when no spares available *)
      ENDIF;
   ENDIF;

   DEATHIF 2 * NFP[J] >= NP[J] AND NP[J] > 0;
      (* Two faults in an active triad is system failure *)

ENDFOR;

DEATHIF NT = 0;  (* System failure by exhaustion *)
```

Since variable-sized arrays were used, a system with any number of initial triads may be modeled by setting the constant N_TRIADS. As in the example above, the number of spares initially is set with the constant N_SPARES. Table 1 shows that changing these two constants has a dramatic effect on the number of states in the model generated.

		Number of Spares			
		0	1	2	3
Number	1	4	10	19	31
of	2	45	61	85	117
Triads	3	219	259	319	399
	4	889	985	1129	1321

Table 1. Number of states in model for various initial configurations of example 2.

Example 3. Quad With Transient Faults

In this example, a quad architecture with both permanent and transient faults is modeled. The system behavior is as follows:

1. Permanent faults arrive at rate λ.
2. Transient faults arrive at rate θ.
3. Transient faults disappear at rate W.
4. Reconfiguration of processors with permanent faults or transient faults that remain long enough to be detected occurs at rate δ.
5. System failure occurs when a majority of the processors have permanent or transient faults.

The ASSIST input file to describe this system is as follows:

```
NP = 4;                          (* Number of processors *)
LAMBDA = 1E-4;                   (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;               (* Transient fault arrival rate *)
W = .5;                          (* Transient fault disappearance rate *)
DELTA = 3.6E3;                   (* Reconfiguration rate *)

SPACE = (NW: 0..NP,              (* Number of working processors *)
         NFP: 0..NP,             (* Active procs. with permanent faults *)
         NFT: 0..NP);            (* Active procs. with transient faults *)
START = (NP, 0, 0);              (* Start with 4 non-faulty processors *)

DEATHIF NFP+NFT >= NW;           (* Majority of active processors failed *)

IF NW>0 THEN
   TRANTO (NW-1, NFP+1, NFT) BY NW*LAMBDA;  (* Permanent fault arrival *)
   TRANTO (NW-1, NFP, NFT+1) BY NW*GAMMA;   (* Transient fault arrival *)
ENDIF;

IF NFT > 0 THEN
   TRANTO (NW+1, NFP, NFT-1) BY FAST W;
      (* Transient fault disappearance *)
   TRANTO (NW, NFP, NFT-1) BY FAST DELTA;
      (* Transient fault reconfiguration *)
ENDIF;

IF NFP>0 TRANTO (NW, NFP-1, NFT) BY FAST DELTA;
   (* Permanent fault reconfiguration *)
```

The model generated for this example contains 15 states and 20 transitions. This ASSIST file could be used to model a triad, a quintet, or any number of starting processors by changing the constant NP. With 7 initial processors, the model contains 50 states and 100 transitions.

Example 4. Monitored Sensor Failure

In this example, a triad of monitored sensors with imperfect coverage of second failures is modeled. The system behavior is as follows:

1. Sensors fail at rate λ_S.
2. Monitors fail at rate λ_M.
3. The first failed sensor is removed with a mean of MEAN_1 and a standard deviation of SD_1.
4. The second failed sensor is removed with a mean of MEAN_2 and a standard deviation of SD_2.
5. Coverage for second failures is .98.
6. The system fails if the majority of sensors fail, or if half of the sensors fail and less than 2 monitors are working.

The ASSIST input file to describe this system is as follows:

```
LAMBDA_S = 1E-5;              (* Failure rate of sensors *)
LAMBDA_M = 1E-6;              (* Failure rate of monitors *)
MEAN_1 = 3E-4;                (* Mean recovery time for 1st failure *)
SD_1 = 1E-4;                  (* S.D. of recovery time for 1st failure *)
MEAN_2 = 1E-4;                (* Mean recovery time for 2nd failure *)
SD_2 = 2E-5;                  (* S.D. of recovery time for 2nd failure *)
COV_2 = .98;                  (* Coverage for second failure *)

SPACE = (NS: 0..3,            (* Number of active sensors *)
         NFS: 0..3,           (* Number of failed active sensors *)
         NM: 0..3);           (* Number of working monitors *)
START = (3, 0, 3);

DEATHIF NFS >= NS;            (* All sensors failed *)
DEATHIF NFS > 1;              (* 2/3 sensors failed *)
DEATHIF NS=2 AND NM<2 AND NFS=1;
          (* 1/2 sensors fail and less than 2 monitors working *)

IF NS>0 TRANTO NFS = NFS+1 BY (NS-NFS)*LAMBDA_S;   (* Sensor failure *)
IF NM>1 TRANTO NM = NM-1 BY NM*LAMBDA_M;           (* Monitor failure *)
```

```
   (* First failure recovery *)
IF NS>2 AND NFS>0 THEN
   IF NM>1 TRANTO (NS-1, NFS-1, NM-1) BY <MEAN_1,SD_1,(NM/NS)>
         (* Loss of monitored sensor *)
   IF NM>1 AND NS>NM TRANTO (NS-1, NFS-1, NM) BY <MEAN_1,SD_1,(NS-NM)/NS)>
         (* Loss of unmonitored sensor *)
   IF NM<2 TRANTO (NS-1, NFS-1, NM) BY <MEAN_1,SD_1>
         (* Loss of either of 2 unmonitored sensors *)
ENDIF;

   (* Second failure recovery *)
IF NS=2 AND NM=2 AND NFS>0 THEN
   TRANTO (NS-1, NFS-1, NM) BY <MEAN_2,SD_2,COV_2>
         (* Successfully removed failed sensor *)
   TRANTO (NS-1, NFS, NM) BY <MEAN_2,SD_2,1.0-COV_2>
         (* Mistakenly removed nonfaulty sensor *)
ENDIF;
```

The semi-Markov model generated for this example contains 18 states and 24 transitions.

Example 5. Two Triads with Three Power Supplies

This example consists of two triads of computers with one triad of power supplies connected such that one computer in each triad is connected to each power supply. Thus, if a power supply fails, then one computer in each triad fails. Because of the complex failure dependencies, this is not an easy system to model. The usual method of using state space variables to represent the number of failed computers in each triad is insufficient because which computers have failed is also important state information. One way to model this system is to use the state space variables as flags to indicate the failure of each computer and power supply in the system. This uses a large number of state space variables, but the system can be described using only a few simple TRANTO statements. The large number of state space variables, however, leads to an unnecessarily complex semi-Markov model. The ASSIST input file is as follows:

```
LAM_PS = 1E-6;                (* Failure rate of power supplies *)
LAM_C = 1E-5;                 (* Failure rate of computers *)

SPACE = (CAF: ARRAY[1..3] OF 0..1,      (* Failed computers in Triad A *)
         CBF: ARRAY[1..3] OF 0..1,      (* Failed computers in Triad B *)
         PSF: ARRAY[1..3] OF 0..1);     (* Failed power supplies *)
START = (9 OF 0);

DEATHIF CAF[1] + CAF[2] + CAF[3] > 1;   (* 2/3 computers in Triad A failed *)
DEATHIF CBF[1] + CBF[2] + CBF[3] > 1;   (* 2/3 computers in Triad B failed *)

FOR I = 1,3
   IF CAF[I]=0 TRANTO CAF[I] = 1 BY LAM_C;
      (* Failure of computer in Triad A *)
   IF CBF[I]=0 TRANTO CBF[I] = 1 BY LAM_C;
      (* Failure of computer in Triad B *)
   IF PSF[I]=0 TRANTO CAF[I] = 1, CBF[I] = 1, PSF[I] = 1 BY LAM_PS;
      (* Power supply failure *)
ENDFOR;
```

This rather brute-force method of modeling the system leads to a semi-Markov model with 70 states and 138 transitions to model this relatively simple system.

As can be seen from this example, modeling of systems using semi-Markov models is still rather much of an art, even using the ASSIST program. As with any language, once the user becomes proficient in using the ASSIST language, he can more easily see how to generate more elegant models. Often, a model can be made considerably smaller by using fewer state space variables to describe the system states, although this sometimes leads to rather complex TRANTO and DEATHIF statements. Using state space variables to represent the number of failed computers in each triad and adding a flag to signal the dependencies between failed computers, the system may be modeled with a much smaller state space. Combining the resulting complex transition rules by logical reasoning, the system described above can be modeled by the following input file:

```
LAM_PS = 1E-6;                (* Failure rate of power supplies *)
LAM_C = 1E-5;                 (* Failure rate of computers *)

SPACE = (NFP: ARRAY[1..2] OF 0..3,  (* Number of failed          *)
                                    (*    computers in each triad *)
         NFS: 0..3,                 (* Number of failed power supplies *)
         SAME: 0..1);  (* Set to 0 if 2 failed computers are on    *)
                       (*    different power supplies, 1 otherwise *)
START = (0, 0, 0, 1);

DEATHIF NFP[1]>1 OR NFP[2]>1;
   (* The system fails if 2/3 computers in either triad fail *)

FOR I=1,2
   IF NFP[I]<3 THEN
      IF NFP[3-I]=1 THEN   (* Other triad has a failed computer *)
         TRANTO NFP[I] = NFP[I]+1 BY LAM_C;
            (* Failure of computer on same power supply as other failed one *)
         TRANTO NFP[I] = NFP[I]+1, SAME = 0 BY (2-NFP[I])*LAM_C;
            (* Failures of computers on different power *)
            (* supplies than the other failed one       *)
      ELSE
         TRANTO NFP[I] = NFP[I]+1 BY (3-NFP[I])*LAM_C;
            (* Failures of computers when other triad has no failures yet *)
      ENDIF;
   ENDIF;
ENDFOR;
```

```
IF (NFP[1]=0 AND NFP[2]=0) THEN
   TRANTO (NFP[1]+1, NFP[2]+1, NFS+1, 1) BY 3*LAM_PS;
      (* Power supply failures when no previous *)
      (* computer failures have occurred.       *)
ELSE
   TRANTO (2, 2, 2, 0) BY (3-SAME)*LAM_PS;
      (* Failure of a power supply not connected to another  *)
      (* previously failed computer.  NOTE: State (2,2,2,1)  *)
      (* is an aggregation of several death states.          *)
   IF SAME = 1 TRANTO (1, 1, 1, 1) BY *LAM_PS;
      (* Failed power supply connected to *)
      (* a previously failed computer.    *)
ENDIF;
```

This second ASSIST input file leads to a semi-Markov model with only 17 states and 30 transitions to model the same system that using the first strategy required 70 states and 138 transitions. However, this input file is much more difficult to understand and verify.

CONCLUDING REMARKS

The use of the ASSIST program has been described and illustrated by several examples. This program allows the user to define a model in an abstract semi-Markov model definition language which can be used to specify reliability models. The language essentially defines a set of rules which are used to automatically generate the semi-Markov model. 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. Furthermore, 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 could represent a large increase in the size of the generated semi-Markov model.

APPENDIX A. ERROR MESSAGES

The following error messages are generated by the ASSIST program. These are listed in alphabetical order:

ARGUMENT TO STANDARD FUNCTION MISSING - No argument was supplied for a standard function.

BOOLEAN EXPRESSION REQUIRED - The condition expression of a TRANTO statement or a DEATHIF statement must be a boolean expression.

BY EXPECTED - The BY keyword was expected.

CANNOT ACCESS A FOR VARIABLE OUTSIDE ITS LOOP - A FOR loop variable is only defined in statements between the FOR statement and the matching ENDFOR statement.

COMMA EXPECTED - Syntax error; a comma is needed.

COMMENT NEVER TERMINATED - The end of the ASSIST input file was reached before a comment was terminated.

CONSTANT EXPECTED - Syntax error; a constant is needed.

DIMENSIONS OF A STATE MUST BE ___ - The dimensions of the state specified in the START statement or in the destination expression of a TRANTO statement must be the same dimensions as in the SPACE statement.

DIVISION BY ZERO NOT ALLOWED - A division by zero was encountered when evaluating the expression.

ENDIF EXPECTED - The ENDIF keyword is expected.

ERROR CREATING VMS LISTING FILE - ASSIST was unable to create the listing file.

ERROR CREATING VMS MODEL FILE - ASSIST was unable to open the model file.

FILE NAME EXPECTED - Syntax error; the file name is missing.

FILE NAME TOO LONG - File names must be 80 or less characters.

IDENTIFIER EXPECTED - Syntax error, an identifier is needed.

IDENTIFIER TOO LONG - Only 8 characters allowed in an identifier name.

IDENTIFIER NOT DEFINED - The identifier used has not yet been defined.

ILLEGAL CHARACTER - The character used is not recognized by ASSIST.

ILLEGAL STATEMENT - The command word used is not recognized by ASSIST.

INPUT LINE TOO LONG - The command line exceeds the 100 character limit.

INTEGER EXPECTED - Syntax error; an integer is needed.

INTEGER OR REAL EXPECTED - Syntax error; an integer or a real number is needed.

LOWER BOUND OF FOR MUST BE <= UPPER BOUND - The lower bound specified for a FOR loop variable must be less than or equal to the upper bound specified.

NO MATCHING IF - An ELSE or an ENDIF was encountered outside of an IF statement.

NUMBER TOO LONG - Only 15 digits/characters allowed per number.

REAL EXPECTED - Syntax error; a real or floating point number is needed.

SEMICOLON EXPECTED - Syntax error; a semicolon is needed.

SPACE DIMENSIONS NOT DEFINED YET - The SPACE statement must precede the START statement or any TRANTO or DEATHIF statements.

SPECIFICATION ERRORS - MODEL NOT GENERATED - One or more errors were encountered during parsing, and model generation will not be attempted.

START STATE NOT DEFINED YET - MODEL NOT GENERATED - The START statement was not found in the ASSIST input file, and model generation will not be attempted.

START STATE NOT WITHIN DEFINED SPACE - One or more state space variable values specified in the START statement are not within the bounds set for that variable in the SPACE statement.

STATE VARIABLE EXPECTED - Syntax error; a state variable is needed.

SUB-EXPRESSION TOO LARGE, i.e. > 1.70000E+38 - An overflow condition was encountered when evaluating the expression.

TRANTO EXPECTED - The TRANTO keyword is expected.

VMS FILE NOT FOUND - The ASSIST input file specified is not present on the disk.

.. EXPECTED - Syntax error; the .. operator is needed.

= EXPECTED - Syntax error; the = operator is needed. ASSIST is parsing either a constant definition statement or assigned values in a TRANTO destination state expression.

) EXPECTED - A right parenthesis is missing in the expression.

(EXPECTED - A left parenthesis was expected.

(OR STATE VARIABLE EXPECTED - The destination of the TRANTO was expected.

] EXPECTED - A right square bracket is missing in the expression.

[EXPECTED - The array index of an array state space variable must be specified.

APPENDIX B. EXAMPLE MODEL AND LISTING FILES

This appendix contains the model file and listing file generated by ASSIST for example problem 1.

The model file describes the semi-Markov model generated in the form needed for input to the SURE program. The first 5 lines contain all of the constant definitions used in ASSIST. The remainder of the file contains all of the state transitions defining the model. SURE assumes that state 1 is the start state of the model, and each state with no transitions out of it is a death state. The states are identified only as a single integer in SURE; however, the state space vector used by ASSIST to identify each state is included in comments for the user.

```
N_PROCS =  3;
N_SPARES =  2;
LAMBDA_P =  1E-4;
LAMBDA_S =  1E-5;
DELTA =  3.6E3;
   1(* 3,0,2,0 *),   2(* 3,1,2,0 *) = (3-0)*LAMBDA_P;
   1(* 3,0,2,0 *),   3(* 3,0,2,1 *) = 2*LAMBDA_S;
   2(* 3,1,2,0 *),   4(* 3,2,2,0 *) = (3-1)*LAMBDA_P;
   2(* 3,1,2,0 *),   5(* 3,1,2,1 *) = 2*LAMBDA_S;
   2(* 3,1,2,0 *),   6(* 3,0,1,0 *) = (1-(0/2))*1*DELTA;
   3(* 3,0,2,1 *),   5(* 3,1,2,1 *) = (3-0)*LAMBDA_P;
   3(* 3,0,2,1 *),   7(* 3,0,2,2 *) = 2*LAMBDA_S;
   5(* 3,1,2,1 *),   8(* 3,2,2,1 *) = (3-1)*LAMBDA_P;
   5(* 3,1,2,1 *),   9(* 3,1,2,2 *) = 2*LAMBDA_S;
   5(* 3,1,2,1 *),  10(* 3,0,1,1 *) = (1-(1/2))*1*DELTA;
   5(* 3,1,2,1 *),  11(* 3,1,1,0 *) = (1/2)*1*DELTA;
   6(* 3,0,1,0 *),  11(* 3,1,1,0 *) = (3-0)*LAMBDA_P;
   6(* 3,0,1,0 *),  10(* 3,0,1,1 *) = 1*LAMBDA_S;
   7(* 3,0,2,2 *),   9(* 3,1,2,2 *) = (3-0)*LAMBDA_P;
   9(* 3,1,2,2 *),  12(* 3,2,2,2 *) = (3-1)*LAMBDA_P;
   9(* 3,1,2,2 *),  13(* 3,1,1,1 *) = (2/2)*1*DELTA;
  10(* 3,0,1,1 *),  13(* 3,1,1,1 *) = (3-0)*LAMBDA_P;
  11(* 3,1,1,0 *),  14(* 3,2,1,0 *) = (3-1)*LAMBDA_P;
  11(* 3,1,1,0 *),  13(* 3,1,1,1 *) = 1*LAMBDA_S;
  11(* 3,1,1,0 *),  15(* 3,0,0,0 *) = (1-(0/1))*1*DELTA;
  13(* 3,1,1,1 *),  16(* 3,2,1,1 *) = (3-1)*LAMBDA_P;
  13(* 3,1,1,1 *),  17(* 3,1,0,0 *) = (1/1)*1*DELTA;
  15(* 3,0,0,0 *),  17(* 3,1,0,0 *) = (3-0)*LAMBDA_P;
  17(* 3,1,0,0 *),  18(* 3,2,0,0 *) = (3-1)*LAMBDA_P;
```

The listing file is created to help the user verify that the model generated describes the fault and recovery behaviors intended. The name of the associated ASSIST input file and the model file are printed. Then, a listing of the ASSIST input file is printed with line numbers. If there were any errors during parsing of the input file, they would be printed in the listing. Also, any error or warnings encountered during model generation are printed in the listing file. The STATE TRANSITIONS section contains a list of the destination state of each transition leaving each non-death state in the model. Destination states that are death states are annotated with an asterisk. The SURE STATE MAPPINGS section contains a list of the mappings between the SURE state numbers and the state variables of that state in ASSIST. The last lines contain the model statistics. The statistics printed are: the processing time (the execution time for generating the model after parsing of the input file was completed), the number of states in the model, and the number of transitions in the model.

```
  INPUT FILE NAME: MANUAL1.AST
  MODEL FILE NAME: MANUAL1.MOD

 1: (*  TRIAD WITH COLD SPARES  *)
 2:
 3: N_PROCS = 3;              (* Number of active processors *)
 4: N_SPARES = 2;             (* Number of spare processors *)
 5: LAMBDA_P = 1E-4;          (* Failure rate of active processors *)
 6: LAMBDA_S = 1E-5;          (* Failure rate of spare processors *)
 7: DELTA = 3.6E3;            (* Reconfiguration rate *)
 8:
 9: SPACE = (NP: 0..N_PROCS,         (* Number of active processors *)
10:          NFP: 0..N_PROCS,       (* Number of failed active processors *)
11:          NS: 0..N_SPARES,       (* Number of spare processors *)
12:          NFS: 0..N_SPARES);     (* Number of failed spare processors *)
13:
14: START = (N_PROCS, 0, N_SPARES, 0);
15:
16: DEATHIF 2 * NFP >= NP;
17:
18: IF NP > NFP TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_P;
19:    (* Active processor failure *)
20:
21: IF NS > NFS TRANTO NFS = NFS+1 BY NS*LAMBDA_S;
22:    (* Spare processor failure *)
23:
```

```
24: IF (NFP > 0 AND NS > 0) THEN
25:    IF NS > NFS TRANTO (NP, NFP-1, NS-1, NFS) BY (1-(NFS/NS))*NFP*DELTA;
26:       (* Replace failed processor with working spare *)
27:
28:    IF NFS > 0 TRANTO (NP, NFP, NS-1, NFS-1) BY (NFS/NS)*NFP*DELTA;
29:       (* Replace failed processor with failed spare *)
30: ENDIF;
```

```
STATE TRANSITIONS:
( 3, 0, 2, 0) -> ( 3, 1, 2, 0) , ( 3, 0, 2, 1)
( 3, 1, 2, 0) -> ( 3, 2, 2, 0)*, ( 3, 1, 2, 1) , ( 3, 0, 1, 0)
( 3, 0, 2, 1) -> ( 3, 1, 2, 1) , ( 3, 0, 2, 2)
( 3, 1, 2, 1) -> ( 3, 2, 2, 1)*, ( 3, 1, 2, 2) , ( 3, 0, 1, 1) ,
                 ( 3, 1, 1, 0)
( 3, 0, 1, 0) -> ( 3, 1, 1, 0) , ( 3, 0, 1, 1)
( 3, 0, 2, 2) -> ( 3, 1, 2, 2)
( 3, 1, 2, 2) -> ( 3, 2, 2, 2)*, ( 3, 1, 1, 1)
( 3, 0, 1, 1) -> ( 3, 1, 1, 1)
( 3, 1, 1, 0) -> ( 3, 2, 1, 0)*, ( 3, 1, 1, 1) , ( 3, 0, 0, 0)
( 3, 1, 1, 1) -> ( 3, 2, 1, 1)*, ( 3, 1, 0, 0)
( 3, 0, 0, 0) -> ( 3, 1, 0, 0)
( 3, 1, 0, 0) -> ( 3, 2, 0, 0)*
```

```
SURE STATE MAPPINGS:
STATE 1 = (3, 0, 2, 0)
STATE 2 = (3, 1, 2, 0)
STATE 3 = (3, 0, 2, 1)
STATE 4 = (3, 2, 2, 0)
STATE 5 = (3, 1, 2, 1)
STATE 6 = (3, 0, 1, 0)
STATE 7 = (3, 0, 2, 2)
STATE 8 = (3, 2, 2, 1)
STATE 9 = (3, 1, 2, 2)
STATE 10 = (3, 0, 1, 1)
STATE 11 = (3, 1, 1, 0)
STATE 12 = (3, 2, 2, 2)
STATE 13 = (3, 1, 1, 1)
STATE 14 = (3, 2, 1, 0)
STATE 15 = (3, 0, 0, 0)
STATE 16 = (3, 2, 1, 1)
STATE 17 = (3, 1, 0, 0)
STATE 18 = (3, 2, 0, 0)

PROCESSING TIME =  1.15
NUMBER OF STATES IN MODEL      = 18
NUMBER OF TRANSITIONS IN MODEL = 24
```

REFERENCES

1. White, Allan L.: Upper and Lower Bounds for Semi-Markov Reliability Models of Reconfigurable Systems. NASA CR-172340, 1984.

2. Butler, Ricky W.: The SURE Reliability Analysis Program, NASA TM-87593, February 1986.

3. Bavuso, Salvatore J.: A User's View of CARE III. 1984 Annual Reliability and Maintainability Symposium, January 1984.

4. Trivedi, Kishor; Geist, Robert; Smotherman, Mark; and Dugan, Joanne Bechta: Hybrid Modeling of Fault-Tolerant Systems. Computers and Electrical Engineering, An International Journal, vol. 11, no. 2 and 3, pp. 87-108, 1985.

5. Butler, Ricky W.: An Abstract Specification Language for Markov Reliability Models, NASA TM-86423, April 1985.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-87735		

4. Title and Subtitle	5. Report Date
Assist User's Manual	August 1986
	6. Performing Organization Code: 505-66-21-01
7. Author(s): Sally C. Johnson	8. Performing Organization Report No.
	10. Work Unit No.
9. Performing Organization Name and Address: NASA Langley Research Center, Hampton, Virginia 23665	11. Contract or Grant No.
	13. Type of Report and Period Covered: Technical Memorandum
12. Sponsoring Agency Name and Address: National Aeronautics and Space Administration, Washington, D. C. 20546	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

Semi-Markov models can be used to compute 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 devastingly tedious and error-prone. The ASSIST program allows the user to describe the semi-Markov model in a high-level language. Instead of specifying the individual states of the model, the user specifies the rules governing the behavior of the system and these are used by ASSIST to automatically generate the model. The ASSIST program is described and illustrated by examples.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Reliability Analysis Markov Models Reliability Modeling Fault Tolerance	Unclassified - Unlimited Subject Category 65

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	41	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document