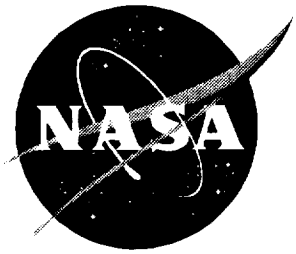
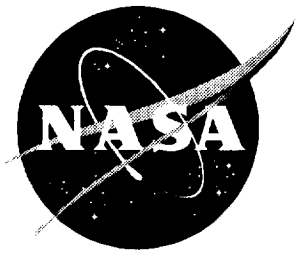
NASA Technical Memorandum 4592



ASSIST User Manual

Sally C. Johnson and David P. Boerschlein

August 1995



NASA Technical Memorandum 4592

ASSIST User Manual

Sally C. Johnson
Langley Research Center • Hampton, Virginia

David P. Boerschlein
Lockheed Engineering & Sciences Company • Hampton, Virginia

August 1995

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available electronically at the following URL address: http://techreports.larc.nasa.gov/ltrs/ltrs.html

Printed copies available from the following:

NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Contents

List of Tables

List of Figures

Abstract

Semi-Markov models can be used to analyze the reliability of virtually any fault-tolerant system. However, the process of delineating all the states and transitions in the model of a complex system can be devastatingly tedious and error prone. The Abstract Semi-Markov Specification Interface to the Semi-Markov Unreliability Range Evaluator (SURE) Tool (ASSIST) computer program allows the user to describe the semi-Markov model in a high-level language. Instead of listing the individual model states, the user specifies the rules governing the behavior of the system, and these are used to generate the model automatically. A few statements in the abstract language can describe a very large, complex model. Because no assumptions are made about the system being modeled, ASSIST can be used to generate models describing the behavior of any system. The ASSIST program and its input language are described and illustrated by examples.

10. 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 Unreliability Range Evaluator (SURE) program, enable the accurate solution of extremely large and complex Markov models (refs. 1 and 2). (In this paper, the term *Markov* will be used to refer to both Markov and the more general semi-Markov models.) 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. Many of the early fault-tolerant architectures are relatively simple to model. Even the early complex systems usually had subsystems that could be modeled independently. However, as flight-critical systems become more complex and more highly integrated, the Markov models describing them will become enormously complex. The complexity of the model stems from the interactions between failure and recovery processes of the various subsystems, which can no longer be modeled independently.

Often, even the most complex system characteristics 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 rules describing the behavior of each subsystem can be developed and verified separately; the submodels are then easily combined to accurately model the behavior of the entire integrated system. Butler (ref. 3) developed an abstract, high-level language for describing system behavior rules and a methodology for automatically generating semi-Markov models from the language. This methodology was implemented in a computer program, the Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST). ASSIST is written in ANSI-standard "C" and executes under the VMS and UNIX operating systems. The ASSIST program produces a file containing the generated semi-Markov model in the format needed for input to the Langley-developed reliability analysis programs SURE, STEM (Scaled Taylor Exponential Matrix), and PAWS (Padé Approximation with Scaling (ref. 4)). For Markov analysis programs requiring a different form of input for the Markov model, a simple program could be written to modify the model description file.

Describing a system in the ASSIST abstract input language forces the reliability engineer to understand clearly the fault tolerance strategies of the system, and the abstract description is also useful for communicating and validating the system model.

This paper describes the ASSIST computer program and input language and shows a few simple examples. For a detailed tutorial on how to use the SURE and ASSIST programs, the reader is referred

to reference 5. The basic concepts of the ASSIST program are introduced in section 2. The syntax of the ASSIST input language is detailed in section 3, and techniques for reducing the size of the generated model are described in section 4. Section 5 presents the algorithm used to generate the semi-Markov reliability model from the input description. The files generated by the program are described in section 6. Section 7 leads the reader through a number of example problems. Finally, the commands for executing the ASSIST program are presented in section 8, followed by the Concluding Remarks.

11. Basic Concepts

The ASSIST program is based on concepts that are used to design compilers. The ASSIST input language defines rules for generating a model. The model states are defined by a set of state-space variables, which represent system-state characteristics such as the number of failed processors or the number of spares. The model generation rules are first applied to a *start state*. The rules create transitions from the start state to new states. The program then applies the rules to the newly created states. This process is continued until all states are either *death* states or have already been processed. The expressiveness of the ASSIST language is derived from the use of *recursive* semantics for its constructs. Thus, a small, compact description in the ASSIST input language can efficiently, yet accurately, represent the failure behavior of a system that may require an extremely large semi-Markov model to solve for system-failure probability.

Absorbing model states (i.e., states with no transitions leaving them) represent system failure. Typically, the reliability engineer needs to determine the probability of entering an absorbing state within the specified mission time. The absorbing state (death conditions) of the model must be defined in terms of state-space variables. These death conditions could be system failure, the onset of degraded performance operation, or other situations resulting from failures.

Consider an example system that has a triad of processors performing the same function. A majority voter is used to detect and resolve any discrepancies between the outputs of the three processors. A semi-Markov reliability model for this example appears in figure 6. In this figure, the triad begins operation in state 1 with three working processors. Processor failures are exponentially distributed at a rate of λ. Since there are three processors that can fail, there is a transition leaving state 1 at a rate of 3λ. After one of the three processors fails, the system is in state 2. In this state, the system is still functional since there are two working processors and one failed one. The failed processor is outvoted by the good processors. After one of the two remaining processors fails at rate λ, the system is in state 3. This state is a failure death state, because there are two failed processors and one good processor and the majority voter can no longer correctly eliminate the erroneous outputs. Death states (absorbing states) are precisely those with no arrows leaving them.

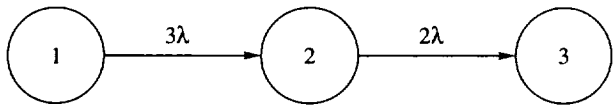


Figure 6. Semi-Markov triad model (SURE state numbers).

An ASSIST input description that would generate the above model is

```
LAMBDA = 1.2E-4;
SPACE = (NP:1..3,NFP:0..3);
START = (3,0);
DEATHIF (NFP>=(NP-NFP));
IF (NFP<3) TRANTO NFP=NFP+1 BY (NP-NFP)*LAMBDA;
```

The first input-description statement defines LAMBDA, a constant with a value of 1.2×10^{-4}. This constant will be used to denote the exponential processor failure rate in the example. The next statement defines the state space as an ordered pair of integers. The first integer is named NP (number of processors) and can take on values between 1 and 3. The second integer is named NFP (number of failed processors) and can take on values between 0 and 3. The third statement indicates that the model starts in state (3,0). This state has three processors and none failed. The words LAMBDA, NP, and NFP are simply names chosen for this particular example; they have no intrinsic meaning to the ASSIST program.

The DEATHIF statement indicates that system failure (death) occurs when the number of failed processors equals or exceeds the number of working processors.

The last statement is a TRANTO statement. The IF clause specifies that this transition is valid if there is still a working processor left. The TRANTO clause describes how to compute the ordered pair representing the new state from the old state values. In this example, the number of faulty processors is increased by one. The BY clause defines the rate at which the transition occurs.

Every ASSIST input description must contain a SPACE statement, a START statement, at least one TRANTO statement, and at least one DEATHIF statement. Figure 7 shows the semi-Markov example model for the system with the states labeled with the corresponding ASSIST state-space values.

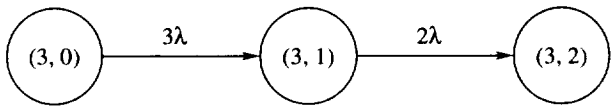


Figure 7. Semi-Markov triad model (ASSIST state numbers).

12. Input-Language Syntax

This section contains a detailed description of the ASSIST input-language syntax. The first subsection introduces the lexical details and basic building blocks of the language, such as identifiers and expressions. The ASSIST statements themselves are then described in the second subsection.

12.1. Language Building Blocks

This section introduces the syntax for the identifier names, numeric and Boolean expressions, and built-in functions that are the basic building blocks of the ASSIST input language.

12.1.1. Identifiers

An identifier is a name that is assigned to a constant or variable in the language. All identifiers must begin with a letter and contain letters, digits, or underscored characters.

The following are valid identifiers:

```
A
ABC
PI
MU_TRIAD
X123
```

An ASSIST identifier must not exceed 28 characters. Identifiers used for named constants that will be passed to SURE must be unique in the first 12 characters. Constant array names should generally not exceed eight characters. Section 3.2.1 (Setup Section) discusses constant definitions and gives more details on the limitations of constant array names.

The following are *invalid* identifiers:

```
6A      (identifiers must begin with a letter)
_X      (identifiers must begin with a letter)
A#2     (special characters are not allowed)
A 3     (embedded whitespace and newlines are illegal)
```

12.1.2. Numeric Expressions

A *numeric expression* is any mathematical expression that produces either an integer or a real number when evaluated. A numeric expression may contain literal values, named constants, state-space variables, index variables, and arithmetic operations.

A literal value, or number, is a contiguous sequence (up to 28 characters) consisting of digits, an optional decimal point, and an optional signed or unsigned exponent that is preceded by the letter e. The ASSIST language requires that each number begin with a digit. The following are legal numbers:

```
6
2.
2.0
1.00001
0.00001
```

The following are *illegal numbers*:

```
1.1.1   (only one decimal allowed)
.1      (does not begin with a digit; use 0.1 instead)
6 3     (embedded blank not allowed within a number)
```

Scientific notation is also allowed. The letter e embedded in a literal numeric value denotes that an integer power of 10 follows. For example, the values of

6.023×10^{23}

4.111×10^{-13}

are written in ASSIST as

```
6.023e23
4.111e-13
```

respectively.

Valid operations in numeric expressions are

```
+     addition
-     subtraction
*     multiplication
/     real division
DIV   integer division for quotient
MOD   integer division for remainder
CYC   integer division for cyclically wrapped quotient
      x CYC y = 1+((x-1) MOD y))
**    exponentiation
( )   parentheses enclose operations to be performed first
```

Operations within parentheses are always performed first. Exponentiation is performed next, with right-to-left associativity. Multiplication and division (including DIV, MOD, and CYC) are performed next, with left-to-right associativity. Addition and subtraction are performed next, with left-to-right associativity. The order of precedence used in ASSIST for evaluating mathematical expressions is discussed in appendix A.

12.1.3. Numeric Precision

The numeric precision of integer arithmetic in ASSIST is at least 32 bits. The maximum integer value is 2147483647. The minimum integer value is –2147483648. Future ports to machines with 60- or 64-bit architectures may allow even larger integers.

The numeric precision of real number arithmetic in ASSIST is at least 12 significant digits. The exponent size range is machine dependent and amounts to double precision on most 32-bit architectures and to single precision on most 60- or 64-bit architectures. Currently, ASSIST is supported only on 32-bit machines.

12.1.4. Boolean Expressions

A *Boolean expression* is any mathematical expression that evaluates to TRUE or FALSE. A Boolean expression may contain numeric expressions plus relational operators and conjunctions.

There are two possible literal truth (Boolean) values: TRUE or FALSE. These values must be spelled out in an ASSIST input file and cannot be abbreviated, even though ASSIST abbreviates them to **T** and **F** in comments when writing the model output file for SURE.

Valid relational operators that can be used in Boolean expressions are

```
<     less than
>     greater than
<=    less than or equal to
>=    greater than or equal to
=     equal to
<>    not equal to
```

Valid comparative operators that can be used in Boolean expressions are

```
AND   true when both sides are true; otherwise false
OR    true when either side is true; otherwise false
XOR   true when only one side is true; otherwise false
NOT   true when right side is false; false when right side is true
```

Optionally, an ampersand can be used instead of the word **AND** and a vertical bar can be used instead of the word **OR**. See the language syntax in appendix B for details on the use of special symbols.

Truth tables for the comparators are given in tables 11 through 4.

Table 11. Truth Table for **AND** Conjunction

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Table 2. Truth Table for **OR** Conjunction

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Table 3. Truth Table for **XOR** Conjunction

XOR	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	FALSE

Table 4. Truth Table for **NOT** Conjunction

$\mathcal{P}$	**NOT** $\mathcal{P}$
FALSE	TRUE
TRUE	FALSE

12.1.5. Arrays

The ASSIST language supports the use of both singly and doubly subscripted arrays. A singly subscripted array is simply an ordered sequence of scalars. A scalar is usually an integer or a real number but could also be a Boolean value. The index number refers to the position in the sequence. For example, index 3 corresponds to the third scalar in the sequence. Consider the array

```
ELE = [ 5, 11, 22, 17, 4, 37, 99, 2 ];
```

The **ELE** array has eight scalars. The third scalar is 22. To reference the third scalar in the array, use **ELE[3]**. The square brackets denote an index or subscript used to reference an individual position or scalar value within the array. Reference to the array name without the bracketed index indicates the whole array.

With the exception of a few built-in array functions, arithmetic can be performed only on scalars, not on arrays. Therefore, the user must always use an index after the array name to indicate which array values to use in the arithmetic expression.

There are some built-in functions that operate on entire arrays. One example is the function `SUM`, which, when given an array name, will return the sum of all array elements. For example,

```
ARR = [ 6, 3, 1 ];
ARRSUM = SUM(ARR);
```

In the above example, the array **ARR** contains three scalars. The scalar constant `ARRSUM` is defined as the sum of all scalars in the array **ARR**. This sum is 10. In this example, the following are numerically equivalent:

```
ARRSUM = 10;
ARRSUM = ARR[1] + ARR[2] + ARR[3];
ARRSUM = SUM(ARR);
```

A doubly subscripted array is similar to a table or a matrix. For example, consider the table

```
17  38  24   12  15
28  11  99   54  37
29  44  66  102  13
```

Two index values are required to denote a scalar in a doubly subscripted table. The first index gives the row, and the second index gives the column. The scalar represented by `TABLE[2,5]` has the value **37**. The first index indicates that the value is in the second row. The second index indicates that the value is in the fifth column.

The above array can be typed into an ASSIST input file as illustrated:

```
TABLE = [
            [ 17,  38,  24,   12,  15 ],
            [ 28,  11,  99,   54,  37 ],
            [ 29,  44,  66,  102,  13 ]
        ];
```

12.1.6. Ranges

A range specifies a contiguous sequence of whole numbers (positive integers). The syntax of a range is

```
<expression> .. <expression>
```

The ellipsis **..** indicates a value range beginning with the left value and ending with the right value, inclusively. Thus, **5..9** specifies the numbers **5,6,7,8,9** whereas **5,9** specifies the numbers **5** and **9**.

12.1.7. Repetition

The ASSIST language allows the use of repetition in several places. For example, instead of typing

```
0,0,0,0,0,0,0,0,0,0,0,0,0,3,3,3,3,3
```

one can merely type

```
13 OF 0, 5 OF 3
```

12.1.8. Built-in Functions

Several mathematical, trigonometric, combinatorial, list, and array functions are available in the ASSIST language. These functions can be used in expressions to compute numeric quantities based upon the parameter values passed to them. For example, the expression

```
TRANTO NWP-- BY SQRT(MU)*LAMBDA;
```

can be written to denote a transition to a new state at rate: $\sqrt{\mu} \cdot \lambda$.

12.1.8.1. Mathematical functions. The mathematical functions are listed in table 5. Example uses of these mathematical functions are

```
FOO = SQRT(FOOBAR)
FOO_POW = EXP(0.5*FOO);
LOGDIFF = LN(FOO-FOOBAR)
ABSFOO_PLUS = 1.0 + ABS(FOO)
```

Table 5. Built-in Math Functions

Function	Parameter type	Result	Description
SQRT	Real	Real	$\sqrt{x}$
EXP	Real	Real	e^x
LN	Real	Real	$\ln x$
ABS	Real	Real	$\|x\|$

12.1.8.2. Trigonometric functions. The trigonometric functions are listed in table 6. Examples of expressions using these trigonometric functions are

```
PI = 4.0 * ARCTAN(1.0);
TWOPI = 2.0*PI;
COSPI = COS(PI);
SINHALFPI = SIN(PI/2.0);
ANG = ARCSIN(0.5);
VAL = SQRT ( (COS(LAMBDA*PI))**2 + (SIN(MU*TWOPI))**2 );
```

Table 6. Built-in Trigonometric Functions

Function	Parameter type	Result	Description
SIN	Real	Real	$\sin x$
COS	Real	Real	$\cos x$
TAN	Real	Real	$\tan x$
ARCSIN	Real	Real	$\sin^{-1} x$
ARCCOS	Real	Real	$\cos^{-1} x$
ARCTAN	Real	Real	$\tan^{-1} x$

12.1.8.3. Combinatorial functions. The combinatorial functions are listed in table 7. Examples of expressions using these combinatorial functions are

```
FOO = COMB(12,5)/FACT(4);
FOOBAR = PERM(12,2);
FOO_BAR = GAM(6.113);
```

Table 7. Built-in Combinatorial Functions

Function	Parameter type	Result	Description
FACT	Int	Int	$n!$
COMB	Int,int	Int	$\frac{n!}{k! \cdot (n-k)!}$
PERM	Int,int	Int	$\frac{n!}{(n-k)!}$
GAM	Real	Real	$\Gamma(x)$

Table 8. Built-in Array/List Functions

Function	Parameter type	Result	Description
SUM	Array	Scalar	$\sum_{i=1}^{\dim x} x_i$
SUM	List	Scalar *Where a, b are scalar*	$\text{SUM}(a, x, b) = (a) + \sum_{i=1}^{\dim x} x_i + (b)$
COUNT	Bool-array	Int	$\sum_{i=1}^{\dim r} \delta(r_i)$; $\delta(r_i) = \begin{cases} 1 & \text{if } r_i = \text{true} \\ 0 & \text{if } r_i = \text{false} \end{cases}$
COUNT	Bool-list	Int *Where p, q are scalar*	$\text{count}(p, r, q) = \delta(p) + \sum_{i=1}^{\dim r} \delta(r_i) + \delta(q)$
MIN	List	Scalar	$y : y \in (\text{list})$ and $y \le y_i \forall y_i \in (\text{list})$
MAX	List	Scalar	$y : y \in (\text{list})$ and $y \ge y_i \forall y_i \in (\text{list})$
ANY	Bool-list	Bool-scalar	(COUNT(list) > 0)
ALL	Bool-list	Bool-scalar	(COUNT(list) = dim list)
SIZE	Array	Scalar	$\text{size}(a) = \dim a$

12.1.8.4. Array/list functions. The array/list functions are listed in table 8. All array/list functions operate on lists of arrays, subarrays, or scalars. For example, if $A1$, $A2$, and $A3$ are arrays, and if X, Y, and Z are scalars,

$$\text{SUM}(A1, X, Y, A2, Z, A3) = \left(\sum_{i=1}^{\dim A1} A1_i\right) + (X) + (Y) + \left(\sum_{i=1}^{\dim A2} A2_i\right) + (Z) + \left(\sum_{i=1}^{\dim A3} A3_i\right)$$

where dim *Ax* is the dimension of *Ax*. Note that the array functions operate on all array elements as if they had each been listed individually. If an array is doubly subscripted, all elements in the array table are operated upon. For example, if *X* is a 3 by 5 array, then

$$\text{SUM(X)} = \left(\sum_{i=1}^{3} \sum_{j=1}^{5} X_{i,j} \right)$$

The asterisk wild card can be used to limit summation within a doubly subscripted array to a single row or column, as in

$$\text{SUM(X[FOO,*])} = \left(\sum_{j=1}^{5} X_{FOO,j} \right)$$

$$\text{SUM(X[*,FOO])} = \left(\sum_{i=1}^{3} X_{i,FOO} \right)$$

The new features in ASSIST revisions 7.0 and higher, namely the combinatorial functions

FACT	(factorial)
COMB	(combinatorial)
PERM	(permulation)
GAM	(gamma)

and the absolute value function (ABS), require revision 7.9 or greater of SURE, STEM, or PAWS in order to solve models using these features. Model files produced with ASSIST 7.0 can be processed with older revisions of SURE, STEM, or PAWS, provided that the new functions are not referenced.

12.2. Language Statements

This section describes the syntax for the statements that make up the ASSIST input language. Certain conventions will be used throughout this manual for introducing the syntax of the ASSIST input language statements:

1. All reserved words will appear in bold as in **FOR** and **IN [1 . . 10]**.
2. Items that are enclosed in angle brackets such as <expression> are expressions that must be supplied by the user.
3. Large braces such as { and } are used to denote constructs that may be omitted or repeated as many times as desired.
4. Large brackets such as [and] are used to denote optional constructs that may be present once or omitted.
5. The suffix *list* denotes a sequence of at least one construct in which each construct is separated by a delimiter, such as a comma.

The ASSIST input file can be subdivided into three parts:

- The *setup* section is first in the input file; it contains definitions and terminates with the `SPACE` statement.
- The *start* section of the file contains more definitions and a single `START` statement; it can also contain `FUNCTION` and `IMPLICIT` definitions.
- The *rule* section of the file begins with the first model generation rule statement, such as a `TRANTO` or `DEATHIF` statement. The rule section of the file must not contain any definitions.

Definitions can be organized in any sequence within the setup and start sections. The only restriction is that an identifier must be defined before it is referenced. Comments may be placed anywhere in the ASSIST input file. Quoted SURE statements may also be placed anywhere in the input file. Because SURE statements can appear in more than one section, they will be described next under the setup section description.

12.2.1. Setup Section

The Setup Section is first in the ASSIST input file and contains definitions and a terminating SPACE statement. The definitions can occur in any sequence, except that an identifier may not be referenced before it is defined. The following sections detail the ASSIST statements that are valid in the setup section of the file.

12.2.1.1. Constant definition statement. A constant definition statement equates an identifier to the value it represents. For example

```
NP = 3;
LAMBDA = 0.0052;
RECOVER = 0.005;
```

These constants are also called *named constants* to distinguish them from *literal values*. The word *constant* is used generally to include both named constants and literal values. In the above example, the name LAMBDA is a *named constant* whereas the value 0.0052 is a *literal value*.

Constants can also be defined in terms of previously defined constants as illustrated:

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

Once defined, a named-constant identifier may be used instead of the value it represents. Constant definitions remain static throughout the execution; thus, once a constant is defined, it cannot be redefined to another value.

The ASSIST language assumes that all named constants are numeric constants. If the expression evaluates to an integer, then the constant will be an integer constant. If the expression evaluates to a real number, the constant will be a real constant. If one of the values in the expression is an integer but contains a decimal point, the constant will be a real constant even though its value might be an integer.

The user must use the word BOOLEAN to specify that the constant will be a Boolean constant. For example

```
FLAG = BOOLEAN NP > 3;
```

Optional parentheses may be added for clarity:

```
FLAG = BOOLEAN (NP > 3);
```

The complete syntax of a constant definition is

<identifier> = **[BOOLEAN]** <definition-clause>

where <definition-clause> defines either a scalar or an array. Its syntax is

<expr>
or
[**[** <#> **OF** **]** <expr> **{** **,** **[** <#> **OF** **]** <expr> **}** **]**
or
ARRAY **(** **[** <#> **OF** **]** <expr> **{** **,** **[** <#> **OF** **]** <expr> **}** **)**

The third form is included for compatibility with prior versions. Doubly subscripted arrays are also allowed via repetition of the second form listed above. The syntax is

[
[[<#> **OF**] <expr> { , [<#> **OF**] <expr> }]
{ , [[<#> **OF**] <expr> { , [<#> **OF**] <expr> }] }
]

All nonBoolean constants are echoed to the model output file.

An ASSIST identifier must not exceed 28 characters. Identifiers used for named constants that will be passed to SURE must be unique in the first 12 characters. Constant array names should not exceed 8 characters because ASSIST automatically generates scalar identifier names to pass to SURE by appending the index value. If the upper bound of a singly subscripted constant array index is no more than 9, then 10 characters can be used for the array name. If the upper bounds of the two doubly subscripted constant array indices exceed 9, then 6 or fewer characters may be required. To illustrate this, consider the arrays:

```
SINGLE = [ 1.2, 4.4, 0.00333 ];
DOUBLE = [
           [ 1.1,  1.2,  1.3 ],
           [ 2.1,  2.2,  2.3 ]
         ];
```

The resulting model file output will contain the lines

```
SINGLE_1 = 1.2;
SINGLE_2 = 4.4;
SINGLE_3 = 0.00333;
DOUBLE_1_1 = 1.1;
DOUBLE_1_2 = 1.2;
DOUBLE_1_3 = 1.3;
DOUBLE_2_1 = 2.1;
DOUBLE_2_2 = 2.2;
DOUBLE_2_3 = 2.3;
```

All these model file identifiers are unique to their first 12 characters. An error message will be printed by ASSIST if two separate constant identifier names are not unique to the first 12 characters.

12.2.1.2. Option definitions. The ASSIST language has some predefined identifiers called options that can have states set to one of three values. The values are on, off, or full; equivalently, they are integer values of zero, one or two.

The syntax of an option definition is

<option-name> <flag-status> ;

where the flag status is one of the following:

OFF
or
ON
or
FULL
or
= 0
or
= 1
or
= 2

The following options are currently defined in the ASSIST language:

- The COMMENT option is used to control whether or not the source and destination state nodes of a transition are written to the model output file as an ordered *n*-tuple in comments. The default is COMMENT ON. If the SPACE statement has a great number of state-space variables, it will take many characters on a line to print the state node. In such instances, ASSIST will force COMMENT OFF, regardless of what the user requested. When COMMENT is ON but is forced OFF because of length, a warning message is generated. To suppress the warning, add a COMMENT OFF statement to the input file. Use of COMMENT FULL is equivalent to COMMENT ON;
- The ECHO option controls whether or not the input file lines are echoed to the standard error file. If set to ON, the input file lines are echoed. If the model file must contain the echo option, the statement must be a SURE statement included in quotes as in "ECHO = 1;". Use of ECHO FULL is equivalent to ECHO ON;
- The ONEDEATH option controls the number of death states in the model. If reset to OFF, then each distinct death state will be enumerated as a separate death state in the SURE model output file. In some models thousands of different death states will result. The default is ONEDEATH ON, which forces the lumping of all death states according to the first DEATHIF statement to which that state conformed. State 1 will contain all death states satisfying the first DEATHIF; state 2 will contain all death states that did not satisfy the first DEATHIF but satisfied the second, and so on. Thus, the probability of system failure caused by each condition specified by a DEATHIF statement will be given by the probability of reaching the corresponding death state. Use of ONEDEATH FULL is equivalent to ONEDEATH ON;
- The TRIM option controls the kind of trimming that is done in the model. Trimming is discussed in section 4.2. The default is TRIM OFF.

Unlike constant definitions, which are written to the model file, option definitions are known only to ASSIST.

12.2.1.3. INPUT statement. An INPUT statement specifies that the user should be queried for the values of one or more named constants.

The syntax is

INPUT <input-list> **;**

where <input-list> consists of a series of identifiers separated by commas and where each identifier is optionally preceded by a prompt message.

For example, interactive input for identifiers LAMBDA and DELTA may be specified as follows:

```
INPUT LAMBDA,DELTA;
```

This statement will result in the default prompt messages during execution:

```
LAMBDA?
DELTA?
```

For the above example, the user is first prompted for LAMBDA and must enter an integer, real value, or expression. The user is then prompted for DELTA and must again enter an integer, value, real value, or expression. Each expression must be entered with a terminating semicolon.

The previous example could also have been specified as follows:

```
INPUT "Enter failure rate of a processor:",LAMBDA,
      "Enter rate to reconfigure in a spare:",DELTA;
```

in which case the prompt messages would read

```
Enter failure rate of a processor:
Enter rate to reconfigure in a spare:
```

More generally, the syntax of an input item in the input list is

<identifier>
or
" <prompt> " : <identifier>

Each entered value or expression must be terminated by a semicolon. Because expressions can span more than one line, pressing the return key is not a substitute for a semicolon. Failure to enter a semicolon will cause ASSIST to wait until one is typed in.

A Boolean constant can also be input. The method is to precede the identifier name with the word BOOLEAN in the INPUT statement. For example,

```
INPUT BOOLEAN FLAG;
INPUT "Enter the flag: " : BOOLEAN FLAG;
```

resulting in respective prompts

```
FLAG?
Enter the flag:
```

12.2.1.4. Comments. Comments in ASSIST must either be initiated with "(*" and terminated with "*)" or be initiated with a left curly bracket "{" and terminated with a right curly bracket "}".

A comment may appear anywhere in the input where an extra space could occur. A comment may not appear in the middle of a literal constant or an identifier. For example, the following are *illegal*:

```
PI = 1.14(* more decimals *)15926;
PI = 1.14{ more decimals }15926;
MU = 4 * P(*comment*)I;
```

When a comment spans more than one line, a capital X appears in the log file for each of the lines that begins with the continuation of a comment. For example,

```
(0001): (*
(0002)X    This is a very very very
(0003)X    long comment.
(0004)X *)
(0005):
```

12.2.1.5. Quoted SURE statement. Statements in the ASSIST input file that are put inside quotation marks are copied into the SURE input file and are not otherwise processed by ASSIST. For example,

```
"INPUT DELTA;"
"FOO = 1 TO 10 BY 2;"
"(* THIS IS A LONG COMMENT TO BE
INCLUDED IN THE SURE INPUT FILE *)"
```

The statements in quotation marks 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 quotation marks in the ASSIST input file. These statements are put in the SURE input file in the sequence encountered. Older versions of ASSIST (before version 7.0), placed these statements before the constant definitions. The new version gives the user more control because constant and option

definitions can now be mixed with quoted SURE statements, and they will be placed in the model output file in the same order in which they were typed into the input file.

12.2.1.6. SPACE statement. The SPACE statement is used to specify the state space on which the model is defined. The state space is defined by an *n*-dimensional vector where each component of the vector is called a *state-space variable* and defines an attribute of the system being modeled. Attributes can be whatever is convenient. Examples might include such things as

- the number of components
- the number of working components
- the number of spare components
- the number of active failed components
- the number with benign faults
- a flag for each component indicating whether it is working

The syntax of the SPACE statement is

SPACE = <space-picture> ;

The syntax of a <space-picture> is

(<space-item> { , <space-item> })

The syntax of a <space-item> is

<ident>
or
<ident> : [**ARRAY** [<expr> . . <expr>] **OF**] <expr> . . <expr>
or
<ident> : [**ARRAY** [<expr> . . <expr>] **OF**] **BOOLEAN**
or
<space-picture>

The ellipsis . . between the square brackets following the word ARRAY denotes the value range over which an index into the array can vary. The ellipses that are shown immediately after the colon for a scalar and immediately after the word OF for an array denote the value range over which the state-space variable can vary, i.e., the value range that legally can be stored for the scalar (array) during the generation phase. Any values between 0 and 32767 are allowed for subscripts or ranges.

The syntax for the SPACE statement is recursive because a space picture can contain a nested space picture within it. This allows for states such as

```
(6,0,(2,2,2),8)
(6,0,(2,2,(4,1,4),TRUE,8),FALSE,4)
```

This feature can be useful for analysis and for checking correctness when a model contains a large number of state-space variables. Long lists of state-space variable values in the model-file comments can be difficult to read unless some subgrouping is done. Consistent nesting of the state space is enforced between the SPACE, START, and TRANTO statements (i.e., if nested parentheses are used in the SPACE statement, they must also be used in the START statement and whenever the entire state space is enumerated in a TRANTO statement). Nested parentheses have no actual effect on the model generation.

Some examples of SPACE statements follow. In these examples, NSI_pool is a constant that must be defined prior to the SPACE statement to represent the *number of spares initially* in the spare pool.

```
SPACE =
  (
   NW_triad  : 0..3, (* Count of working in "triad" *)
   NFA_triad : 0..3, (* Count of active failed in "triad" *)
   NFB_triad : 0..3, (* Count of benign failed in "triad" *)
   NCF       : 0..9  (* Number of component failures for PRUNEIF *)
  );

SPACE =
  (
   (W_triad : ARRAY [0..3] OF BOOLEAN), (* Each working in "triad" *)
   NS_pool : 0..NSI_pool, (* Spares count, COLD pool "pool" *)
   NCF : 0..3+NSI_pool (* Number of component failures for PRUNEIF *)
  );
```

Each variable with an unspecified range has the default range from 0 through 255. Since large models may contain hundreds of thousands of states that must all be stored during rule generation, ASSIST packs the values into as few bits as possible. When explicit ranges are used, only a required number of bits are used, thus greatly decreasing the amount of memory needed to store the model. Thus, execution time speeds up because

- less page swapping of virtual memory is needed
- the state matching (hashing) algorithm is more efficient

Checks are also made during rule generation, and error messages are printed whenever a state is generated that contains a state-space variable that is out of range as defined in the SPACE statement. Thus, for speed and model verification reasons, the user is advised to give each state-space variable an explicit range.

Examples of ranges and corresponding numbers of bits that are required to encode each range are listed in table 9. The number of bits required depends only upon the difference between the upper and lower bounds of a given range, not upon the values themselves.

Table 9. Bits Required to Pack a State-Space Variable

Range	Bits required
BOOLEAN	1
0..1	1
0..2	2
0..3	2
0..4	3
0..7	3
0..8	4
0..15	4
0..16	5
33..34	1
101..116	4

The SPACE statement marks the end of the setup section.

12.2.2. Start Section

The start section of the file contains more definitions plus a single START statement, which can occur anywhere in the section. There must be exactly one START statement or an error message will be printed. Three additional types of macro definitions are allowed in this section, namely the FUNCTION, IMPLICIT, and VARIABLE definitions. The definitions can occur in any sequence, except that an identifier may not be referenced before it is defined.

12.2.3. IMPLICIT Statement

The IMPLICIT statement is used to define a quantity that is not in the state space itself but is a function of the state space. The value of the implicit function is based upon constants and state-space variables.

For example, if NWP is a state-space variable representing the number of working processors, and NI is a constant denoting the number of initial processors, then the declaration

```
IMPLICIT NFP[NWP] = NI - NWP;
```

defines NFP (number of failed processors) to be the difference between the initial number and the current number of working processors. The implicit function shown above can be referenced as illustrated in the following DEATHIF statement:

```
DEATHIF NWP <= NFP;
```

The syntax is

IMPLICIT <identifier> [<state-space-variable-list>] **[** (<parameter-list>) **]** = <body> ;

The <state-space-variable-list> consists of one or more state-space variable-name identifiers separated by commas. The identifiers must already have been defined in a SPACE statement. All state-space variables that are referenced in the body of an IMPLICIT statement must be listed in the state-space variable list.

The optional <parameter-list> is used to declare an IMPLICIT that is also a function of specified parameters. All parameters are positional and must therefore be passed in a consistent order. The user decides how many parameters a certain function will have. The user may not vary the number of parameters that are passed from the number of parameters that are defined. The parameter names in the parameter list are separated by commas. Parameter names may be reused in other IMPLICIT or FUNCTION definitions or as FOR indices, but they cannot be named constants or state-space variable names. Any variables, such as FOR index variables, which are referenced in the body of an IMPLICIT and not already listed as state-space variables, must be listed in the optional parameter list.

The <body> of the IMPLICIT definition is the expression defining the value as a function of the specified state-space variables and parameters. Named constants and reserved words may be freely referenced in the body and must not be specified with either the state-space variables or parameters.

The IMPLICIT may be invoked in TRANTO, DEATHIF, PRUNEIF, ASSERT, IF, or FOR statements or in later IMPLICIT or FUNCTION definitions by giving the name followed by the values in parentheses for each of its parameters. If the IMPLICIT definition does not include a parameter list, the IMPLICIT is invoked by its name alone. The following example illustrates the definition and invocation of IMPLICITs:

```
SPACE = (NWP:ARRAY[1..5] OF 0..3); (* Number working in each of five triads *)

IMPLICIT NFP[NWP](IX) = 3 - NWP[IX]; (* Number failed in each triad *)
IMPLICIT TOTALW[NWP] = SUM(NWP);
IMPLICIT TOTALF[NWP] = 3*5-TOTALW;
```

```
FOR III IN [1..5]
    IF (NFP(III)>0) TRANTO NWP[III]=0 BY 1.E-5;
ENDFOR;
DEATHIF TOTALF > 6;
```

In the above example, the IMPLICIT function NFP is declared a function of the state-space variable NWP and the parameter IX. The value of NFP is computed as three minus the current number of working processors in the triad corresponding to the passed parameter. The IMPLICIT function TOTALW is also a function of the state-space variable NWP but has no parameters. This function value is computed as the number of working processors in all five triads. The IMPLICIT function TOTALF is also a function of the state-space variable NWP but has no parameters. The body of TOTALF references the value of IMPLICIT function TOTALW. Since TOTALW is computed based upon the state-space variable NWP, then NWP must be listed in the state-space variable list for TOTALF, even though it is only referenced indirectly. When NFP is referenced in the IF, the value of III is passed as a parameter to satisfy the parameter requirements as declared in the IMPLICIT NFP line. When TOTALF is referenced in the DEATHIF statement, no parameters are specified none are expected, as was declared in the IMPLICIT TOTALF line.

If an IMPLICIT is declared without a parameter list, then it must be referenced without one. If an IMPLICIT statement is declared with a parameter list, values must be passed for all parameters during invocation so that they can be substituted for their respective parameters. Passed values can be numbers, named constants, variables, or expressions. For example, the following log file excerpt shows the incorrect reference of some IMPLICIT functions:

```
(0005): SPACE = (NWP:ARRAY[1..5] OF 0..3);
(0007): IMPLICIT NFP[NWP](III) = 3 - NWP[III];
(0011):    IF (NFP() < 3) TRANTO NWP[IX]-- BY 1.2e-3;
                  ^ [ERROR] TOO FEW CALLING PARAMETERS.  MORE EXPECTED:  )
                            (IMPLICIT NFP)
(0012):    IF (NFP(IX,IX) < 3) TRANTO NWP[IX]-- BY 1.2e-3;
                    ^ [ERROR] TOO MANY CALLING PARAMETERS. REMAINING IGNORE:  ,
                              (IMPLICIT NFP)
                     ^ [ERROR] SKIPPING EXTRANEOUS TOKENS: IX

(0013):    IF (NFP < 3) TRANTO NWP[IX]-- BY 1.2e-3;
                  ^ [ERROR] LEFT "(" EXPECTED:  < (IMPLICIT NFP)
```

One may wish to define an IMPLICIT that is a function of three FOR index variables. Consider the following example:

```
SPACE =
   (
     AMAT: ARRAY[1..3,1..3] OF 0..7;
     BMAT: ARRAY[1..3,1..3] OF 0..7;
     NCF:0..7    (* Number of component failures for prune *)
   );
IMPLICIT CMAT[AMAT,BMAT](I,J,K) =  AMAT[I,J] * BMAT[J,K];
```

12.2.4. FUNCTION Statement

The FUNCTION statement is used to define a new function. A function computes a value based upon constants and parameters passed to the function. The syntax of a function definition follows:

FUNCTION <identifier> (<parameter-list>) = <body> ;

The <parameter-list> is made up of zero or more identifiers separated by commas. All parameters are positional and must therefore be passed in a consistent order. The user decides how many parameters a certain function will have. The user may not vary the number of parameters that are passed from the

number of parameters that are defined. The parameter names in the parameter list are separated by commas. Parameter names may be reused in other IMPLICIT or FUNCTION definitions or as FOR indices, but they cannot be named constants or state-space variable names.

The <body> of the FUNCTION definition is the expression that defines the value as a function of the parameters.

The FUNCTION may be invoked in TRANTO, DEATHIF, PRUNEIF, ASSERT, IF, or FOR statements or in later IMPLICIT or FUNCTION definitions by giving its name followed by the values for each of its parameters in parentheses.

The following example illustrates the definition and invocation of FUNCTIONs:

```
FUNCTION F(X,Y) = X + Y;
...
IF  2*F(MMM,3)  >  4  THEN
```

In the above example, FUNCTION F is declared to be a function of two parameters, namely X and Y. Its value is computed as the sum of both parameters. The reference to the function F(X,Y) would be expanded to form the following IF clause:

```
IF  2*(MMM+3)  >  4  THEN
```

In this example the expanded form is shown with parentheses around the function operations. When the expression is evaluated and the function is expanded, the operations within the function will be given precedence and will be performed first.

A value must be passed for each parameter during invocation so that it can be substituted for the parameter. A passed value can be a number, a named constant, a variable, or an expression.

The function body may not contain any variables that are not listed in the declared parameter list. For example, if NWP is a state-space variable defined in the SPACE statement, it cannot appear in the body of the FUNCTION. FOR index variables must be listed in the parameter list in order to be referenced in the body. Named constants are allowed in the body of a FUNCTION.

12.2.5. VARIABLE Statement

The VARIABLE statement is used to define a variable that is not explicitly a state-space variable itself but which is dependent upon the state space. The variable can be either a scalar or an array. Variables can be used to make an ASSIST description more understandable without incurring the additional memory required to store extra state-space variable values.

For example, if NWP is a state-space variable representing the number of working processors and NI is a constant denoting the number of initial processors, the declaration

```
VARIABLE NFP[NWP] = NI - NWP;
```

defines NFP, denoting the number of failed processors, to be the difference between the initial number and the current number of working processors. The above variable can be referenced as illustrated in the following DEATHIF statement:

```
DEATHIF NWP <= NFP;
```

VARIABLES and IMPLICITS often produce the same results. The main difference between VARIABLES and IMPLICITS depends upon when the expression evaluation is performed. With an IMPLICIT, the expression to the right of the equals sign is not evaluated until it is actually used. With a VARIABLE the expression is evaluated once for each new state, regardless of whether or not the VARIABLE statement is ever referenced. If the expression is going to be referenced inside a loop or used many different times, a VARIABLE will probably be more efficient than an IMPLICIT. If,

however, the references to the expression are all protected by IF tests that hold true for very few states, an IMPLICIT may be more efficient than a VARIABLE.

Another difference is that a VARIABLE can define an array whereas an IMPLICIT cannot.

The syntax for the VARIABLE statement is

VARIABLE <identifier> [<state-space-variable-list>] = [**BOOLEAN**] <definition-clause> ;

where a definition clause defines either a scalar or an array. Its syntax is

<expr>
or
[[<#> **OF**] <expr> { , [<#> **OF**] <expr> }]
or
ARRAY ([<#> **OF**] <expr> { , [<#> **OF**] <expr> })

The <state-space-variable-list> is made up of one or more state-space variable name identifiers separated by commas. The identifiers must already have been defined in a SPACE statement. All state-space variables that are referenced in the definition clause of a VARIABLE must be listed in the state-space variable list.

The <definition-clause> of the VARIABLE definition is the expression defining the VARIABLE as either a variable or an array which is dependent upon the state space. Named constants and reserved words may be freely referenced to the right of the equals sign and must not be specified with the state-space variables.

The VARIABLE may be referenced in TRANTO, DEATHIF, PRUNEIF, ASSERT, IF, or FOR statements or in later VARIABLE, IMPLICIT, or FUNCTION definitions in exactly the same manner as a constant would be referenced. The VARIABLE is referenced in the same way as a constant would be. Square brackets are used when the VARIABLE is an array. The following example illustrates the definition and use of VARIABLEs:

```
SPACE = (NWP:ARRAY[1..5] OF 0..3); (* Number working in each of five triads *)

VARIABLE NFP[NWP] = [
                      3 - NWP[1],
                      3 - NWP[2],
                      3 - NWP[3],
                      3 - NWP[4],
                      3 - NWP[5]
                    ];   (* Number failed in each triad *)
VARIABLE TOTALW[NWP] = SUM(NWP);
VARIABLE TOTALF[NWP] = 3*5-TOTALW;

FOR III IN [1..5]
   IF (NFP[III]>0) TRANTO NWP[III]=0 BY 1.E-5;
ENDFOR;
DEATHIF TOTALF > 6;
```

In the example above, the variable NFP is dependent upon the state-space variable NWP and is an array with five elements. Each element value is computed as three minus the current number of working processors in the triad corresponding to the array index. The variable TOTALW is also dependent upon the state-space variable NWP. Its value is computed as the total number of working processors in all five triads. The variable TOTALF is also dependent upon the state-space variable NWP. The definition of TOTALF references the value of variable TOTALW. Since TOTALW is computed based upon the state-space variable NWP, then NWP must be listed in the state-space variable list for TOTALF even

though it is only referenced indirectly. When NFP is referenced in the IF clause, the value of III is passed as a subscript to the array NFP.

12.2.6. START Statement

The START statement is used to specify the initial state of the system being modeled; i.e., the probability that the system is in this state at time 0 is 1. The state is defined by an *n*-dimension vector of integers or Booleans.

Each ASSIST input file must contain exactly one START statement that corresponds precisely to the SPACE statement in its setup section; i.e., the START statement must include the same number and type (integer versus Boolean) values in the same sequence as specified in the SPACE statement, and they must use identical nesting.

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 BOOLEAN);
START = (2 OF 6, 0, NX OF FALSE);
```

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

The START statement syntax is

START = <space-expression> ;

The syntax of a <space-expression> is

(<space-expr-item> { , <space-expr-item> })

The syntax of a <space-expr-item> is

[<expr> **OF**] <expr>
or
<space-expression>

Like the SPACE statement, the START statement syntax is recursive because a space expression can contain a nested space expression. Nesting allows for states such as the following:

```
(6,0,(2,2,2),8)
(6,0,(2,2,(4,1,4),TRUE,8),FALSE,4)
```

Only constant expressions may appear in a START statement. The statement must not contain any state-space VARIABLES or IMPLICITS.

Some examples of START statements follow:

```
START =
  (
   3,   (* Count of working in "triad" *)
   0,   (* Count of active failed in "triad" *)
   0,   (* Count of benign failed in "triad" *)
   0    (* Number of component failures for PRUNEIF *)
  );

START =
  (
   (3 OF TRUE), (* Each working in "triad" *)
   NSI_pool,    (* Spares count, COLD pool "pool" *)
```

```
    0              (* Number of component failures for PRUNEIF *)
  );
```

These START statements correspond precisely to the respective examples in the SPACE statement section.

12.2.7. Rule Section

Rules are used to describe the manner in which system components fail and how the system responds to any failures that occur. A typical system has a number of different components that fail at different rates. Some components may have transient or intermittent faults. Some components may be replaced with spares upon failure. There may be dependencies between the components. All these behaviors are formulated into a set of model generation rules.

The rule section of the input file begins with the first rule. Once the rule section begins, definitions are no longer allowed. Several types of statements are valid in the rule section. These are described below.

12.2.7.1. TRANTO statement. The TRANTO statement is the heart of the model generation process and is used to describe the state transitions in the model. The model is generated by recursively applying the TRANTO rules to each model state, beginning with the START state.

The transitions represent the elapsed time between system states, which are stochastic processes defined by probability distributions. In the restricted class of Markov models, all transitions are exponentially distributed and are completely defined by a simple rate parameter. In the more general semi-Markov model, any distribution can be used to describe the elapsed time. Transitions between model states are specified using TRANTO statements. Typical TRANTO statements have three main parts or clauses:

1. An optional IF conditional expression (also called the "IF clause")
2. A required destination expression, (also called the "TRANTO clause" or the "destination clause")
3. A required rate expression, (also called the "BY clause" or "rate clause")

The optional first clause contains a Boolean expression that describes the states for which the transition is valid in terms of their state-space variable values. The second clause defines the destination state for the transition in terms of state-space variable values. The third clause defines the distribution of elapsed time for the transition.

The syntax for the simple TRANTO statement is given by

[IF <condition> **] TRANTO** <dest> **by** <rate> **;**

The condition is a Boolean expression. The destination state can be specified as either a state node (vector format) or as an assignment list (list format). Mixing of the two specification formats in the same TRANTO is not allowed.

State-space variables may be referenced in any of the three clauses of a TRANTO statement, but they can be changed only in the destination clause. Integers can appear in all three clauses, but only the rate clause can contain real numbers.

When a transition is processed, the source-state value for a state-space variable is used if its value does not change during the transition. For example, if NWP and QQQ are state-space variables and NWP=4 and QQQ=22 before the transition, then the clause

```
TRANTO NWP++,QQQ=QQQ-NWP
```

will set NWP to 5 and QQQ to 22-4=18 in the destination state. The source-state value of NWP is used to compute QQQ even though NWP has already been incremented in the destination state during the previous assignment.

Condition expression. The optional first expression following the IF is the condition expression. Condition expressions must be Boolean expressions. Conceptually, the condition expression determines on which model states to apply this rule. For example, in

```
SPACE = (A1: 1..5, A2: 0..1)
...
IF (A1 > 3) AND (A2 = 0) TRANTO ...
```

the conditional expression is true for states (4,0) and (5,0) only. This TRANTO rule will therefore only be applied to these two states.

Note that the key word IF is used to begin two different kinds of statements in the ASSIST language. If the <condition> is followed by the key word TRANTO, as in the above syntax, the statement is a TRANTO statement. If the <condition> is followed by the key word THEN, the statement is a block IF construct and not just a simple TRANTO statement. The simple TRANTO statement contains exactly one transition and may not contain any other rules, such as FOR, DEATHIF statements and so on. The block IF can contain any or all of these and will be detailed in section 12.2.7.2.

Destination state. The vector or assignment list following the TRANTO reserved word defines the transition destination state to be added to the model. The destination state can be specified using positional or assigned values.

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

The syntax of the TRANTO destination is ·

<assignment> { , <assignment> }

or

([<space-expr-item>] { , [<space-expr-item>] })

The syntax of the second form is almost identical to that of the START state. The only difference is that empty positions are legal for the TRANTO but not for the START state.

The syntax of a space expression item is

[<expr> **OF**] <expr>

or

<space-expression>

Positional and assigned descriptions cannot be mixed in the same statement. When assigned values are used, the parentheses are not used, and state-space variables that do not change values need not be specified. For example, if the space is

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

all the following TRANTO statements are equivalent:

```
IF NF[2]>0 TRANTO ( NP-1 , NF[1] , NF[2]-1 , NF[3] , NX ) BY LAMBDA;
IF NF[2]>0 TRANTO ( NP-1 , , NF[2]-1 , , ) BY LAMBDA;
IF NF[2]>0 TRANTO NP=NP-1 , NF[2]=NF[2]-1 BY LAMBDA;
```

Notice that one does not have to specify state-space variables in the destination state when the values do not change. If, however, the positional form is being used, then one must be careful to make sure that the correct number of commas separate the variables being changed. Spaces around the punctuation are not necessary but have been added for clarity.

Because it is so common to increase or to decrease the value of a state-space variable by one, the ++ and −− operations, which add one and subtract one, respectively, are allowed in the destination field. Consider the following two forms, which have the same effect:

```
TRANTO NWP-NWP+1

TRANTO NWP++
```

In addition to providing a convenient shorthand notation, the ++ and −− constructs are actually more efficiently implemented than their longhand counterparts. The first of these requires three operations: one to fetch NFP, one to fetch the number 1, and one to compute the sum. The second requires two operations: one to fetch and one to increment. A seemingly minor speedup of an operation that is performed once for every transition can result in a noticeable speedup in the generation of a very large model.

Only state-space variables may be changed in the destination clause of a TRANTO statement. Because IMPLICIT functions are typically used to define dependent variables that are IMPLICIT functions of the state space, one might be tempted to change them in a destination clause. This practice, however, is not allowed. For example, the following would not be valid:

```
SPACE = (NP:0..5,NWP:0..5);
IMPLICIT NFP(NP,NWP) = NP - NWP;
...
TRANTO NFP++ BY NWP*LAMBDA;
```

The transition should be correctly entered as

```
TRANTO NWP-- BY NWP*LAMBDA;
```

to increment the state-space variable itself rather than the IMPLICIT function name.

Rate expression. The expression following the BY key word indicates the rate of the transition to be added to the model. This expression may contain FOR index variables and state-space variables as well as constants and arithmetic operations. The rate expression is the only expression in which real constants may be used. There are three ways of expressing a rate in the SURE input language that are supported in ASSIST.

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

<expr>

where <expr> is a real expression. 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 the three subexpressions define the following:

<mu> ≡ the conditional mean transition time
<sig> ≡ the conditional standard deviation of the transition time
<frac> ≡ the transition probability

All three parameters are 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 the 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 three rate expressions are all equivalent:

```
BY < 1.0/ALPHA , 1.0/ALPHA > ;
BY < 1.0/ALPHA , 1.0/ALPHA , 1.0 > ;
BY FAST ALPHA;
```

Spaces are not required around the punctuation. They are present for clarity.

12.2.7.2. Block IF construct. The block IF construct at version ASSIST 7.0 relaxes most restrictions from prior versions. Most noticeable is that FOR, DEATHIF, and other rules may now appear inside of block IF's as well as the TRANTO statements and clauses and nested IF's that were allowed in the past. This feature gives the user a lot more flexibility when describing system behavior.

The syntax of a block IF is

IF <condition> **THEN** { <more-rules> } [**ELSE** { <more-rules> }] **ENDIF ;**

Even though a block IF can be typed on a single line as in the example

```
IF NWP > 0 THEN TRANTO NWP--,NCF++ BY NWP*LAMBDA; DEATHIF ...; ENDIF;
```

it is generally clearer to use multiple lines and indentation, as in

```
IF NWP > 0 THEN
   TRANTO NWP--,NCF++ BY NWP*LAMBDA;
   DEATHIF ...;
ENDIF;
```

A semicolon is not necessary after the word THEN in the IF clause. If present, it will be ignored (since the empty statement is also a valid ASSIST statement).

Do not think of the block IF in the procedural sense. Think of it as conditionally selecting a set of rules.

Block IF constructs can be nested with other constructs. For example,

```
 IF (NFP>0) THEN
    FOR III IN [1..3]
 ASSERT (NOT (W[III] AND F[III])); (* either working or failed but not both *)
 IF (NOT W[III]) THEN
    [TRANTO W[III]=TRUE BY FAST PROB*DELTA1; (* disappearance *)
     TRANTO W[III]=TRUE,NS-- BY FAST (1.0-PROB)*DELTA2;  (* reconfiguration *)
    ENDIF;
 ENDFOR;
ENDIF;
```

12.2.7.3. FOR construct. Often several rules are needed that are identical except that they operate on different state-space variables. The different state-space variables can be put in an array, and the FOR construct can be used to define several TRANTO or DEATHIF rules at once. Do not think of the FOR construct as a loop in the procedural sense. Think of it as repeatedly generating a set of rules.

The syntax of a FOR construct is

FOR <index> **IN** <set> { <more-rules> } **ENDFOR ;**

The old FOR construct syntax, which specified a single index range with the lower and upper bounds separated by a comma, is still supported, but the new syntax is recommended for clarity. The formal syntactical description of the old form, therefore, is relegated to the appendix.

A FOR construct generates rules over a set of values. A set in the ASSIST language is a collection of whole numbers. The numbers can be listed individually or specified via ranges.

The syntax of a set is

[<expression> [. . <expression>] { , <expression> [. . <expression>] }]

The ellipsis . . indicates a value range beginning with the left value and ending with the right value, inclusively. Thus, **5..9** specifies the numbers **5,6,7,8,9** whereas **5,9** specifies the numbers **5** and **9**.

Examples of sets are

```
[3]
[1,5]
[1..5]
[1,3,5]
[0..4,6..9]
[1+4*NP..5*NP]
[(4*NP)..(5*NP-1)]
```

Consider a system in which system failure occurs when at least half of any of three types of components are faulty. An array **NA** of range 1 to 3 could represent the number of each type of component active in the system, and an array **NF** of range 1 to 3 could represent the number of each type of component that is faulty and active. The *death* condition could be described by the following:

```
FOR I IN [1..3];
    DEATHIF 2 * NF[I] >= NA[I];
ENDFOR;
```

Thus, in all states of the Markov model where 2 times the value of NF for one of the three components is greater than or equal to **NA** for the same component, system failure occurs.

Even though a FOR can be typed in all cases on a single line as in the example,

```
FOR I IN [1..3] DEATHIF 2 * NF[I] >= NA[I]; ENDFOR;
```

it is generally clearer to use indentation, as in

```
FOR I IN [1..3]
    DEATHIF 2 * NF[I] >= NA[I];
ENDFOR;
```

A semicolon is not necessary after the set in the IN clause. If present, it will be ignored (since the empty statement is also an ASSIST statement).

The FOR variable <index> may be referenced only by statements between the FOR clause and its matching ENDFOR. The same variable name can be used for FOR constructs that follow each other in series; however, a unique variable name must be used for a FOR index that is nested inside another FOR. The rules between the <set> and the ENDFOR (also called the *body* of the FOR) are processed with the index varying over all specified values in the set. FOR constructs may be nested, as in the following:

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

FOR I IN [1..5]
    FOR J IN [1..2]
       IF NC[I] > J TRANTO NF[I] = NF[I]+1 BY J*LAMBDA;
```

```
    ENDFOR;
    DEATHIF NC[I] < NF[I];
ENDFOR;
```

Each ENDFOR matches the most recently preceding FOR. The FOR constructs in the example above generate 10 (5 × 2) TRANTO rules, one for each pair of **(I,J)** values (1,1), (1,2), (2,1), (2,2), ... , (5,2). Five DEATHIF rules are also defined by using values of **I** from 1 to 5. The body of the FOR may contain any valid rules, including block IF's and nested FOR's.

12.2.7.4. ASSERT statement. The ASSERT statement is used to specify a condition that should be true for every model state. During model generation, each model state is checked for compliance with every ASSERT condition. If any ASSERT conditions are violated, the ASSIST program will print a warning to both the terminal and the log file. ASSERT statements are useful for checking that the system described in the ASSIST input file has the intended properties.

The syntax is

ASSERT <condition> **;**

The ASSERT <condition> is a Boolean expression with optional parentheses. For example,

```
ASSERT (NWP + NFP) = NP; (* Check that working processors + failed processors
                            always equal total # processors *)
% ASSERT (NP>0);
% ASSERT ((NP+5)<(NWP-4));
% ASSERT (NP>0) AND (NWP>0) AND (NFP <= 100);
% ASSERT (NP>0) AND (NWP>0) AND (NOT (NFP > 100));
```

12.2.7.5. DEATHIF statement. The DEATHIF statement specifies which states are death states, i.e., absorbing states in the model. The following is an example:

```
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 <condition> **;**

The <condition> is a Boolean expression with optional parentheses. For example

```
DEATHIF (NWP <= NFP); (* death if no longer majority working *)
DEATHIF (2*NWP[I] <= NP[I]) AND (NT=1); (* death if last triad fails *)
```

Unless the ONEDEATH option is turned off explicitly in the input file, generated death states will be aggregated according to which DEATHIF statement they satisfy. See the description of the ONEDEATH option (section 12.2.1.2) for a complete description of this feature.

12.2.7.6. PRUNEIF statement. Because systems with numerous components exhibit a combinatorial explosion of states, the conservative-pruning method can be used to limit the number of model states, and consequently, can greatly decrease the solution time as well as the hardware memory required for solution.

A system model with many components usually has many long paths consisting of one or two failures of each component type before system failure is reached. Because the occurrence of so many failures is unlikely during a short mission, these long paths typically contribute insignificantly to system-failure probability. The dominant system-failure modes are typically the short paths to system failure

consisting of *like* component failures. Model pruning can eliminate the long paths to system failure if one conservatively assumes that system failure occurs earlier on those paths.

The PRUNEIF statement can be used to truncate the Markov model automatically. Assume that the probability of more than **M** component failures occurring during a mission is negligible compared to the probability of system failure. Using NCF as a state-space variable to represent the number of component failures, the following statement appropriately truncates the model:

```
PRUNEIF NCF > M;
```

The model is truncated by assuming system failure when the truncation criteria are met. This method is therefore conservative.

Note that the number of aggregated pruned states reported by ASSIST is actually the number of transitions that resulted in a pruned state, not the number of unique state *n*-tuples that are pruned states.

The PRUNEIF statement specifies which states are pruned states (conservatively assumed to be absorbing states in the model that reduce the total model size). The PRUNEIF statement has the same syntax as the DEATHIF statement:

PRUNEIF <condition> **;**

The <condition> is a Boolean expression with optional parentheses. The alternate spelling "PRUNIF" is also allowed.

Consider the following example:

```
SPACE = (DIM1: 2..4, DIM2: 3..5);
PRUNEIF (DIM1 = 4) OR (DIM2 = 3);
```

This statement defines (4, 3), (4, 4), (4, 5), (2, 3), and (3, 3) as pruned states.

Like the death states, the generated pruned states are lumped together according to the PRUNEIF statement they satisfy. Thus, the contribution to system failure caused by pruning can be determined easily from examining the SURE output. The pruned states are always lumped, regardless of the setting of the ONEDEATH option. In a model that is generated from an input file with three DEATHIF statements and two PRUNEIF statements, states 1 through 3 will be death states corresponding to the three DEATHIF statements, and states 4 and 5 will be pruned states corresponding to the two PRUNEIF statements.

The specific algorithm used to assign state numbers is detailed in section 4.3.

The ASSIST program generates a statement in the model file that identifies the pruned states in the model. For example, the model with three death states and two pruned states would contain the statement

```
PRUNESTATES = (4, 5);
```

The SURE program will use this statement to list the pruned-state probabilities separately from the death state probabilities. (Note: Versions of SURE earlier than 6.0 will simply list the pruned states as if they were death states.)

PRUNEIF statements may be included inside FOR and block IF constructs, but they may not be included within TRANTO statements. These features are described in the next few sections. Using the PRUNEIF statement to reduce model size is discussed and demonstrated in section 4.1.

13. Model Reduction Techniques

Modeling the failure behavior of a complex system with many interacting components can produce a system-state explosion that results in huge models with hundreds of thousands of states that cannot be

held or solved on typical computer systems. This problem has been called the *large state-space problem.* However, prudent use of various model reduction techniques with ASSIST allows the user to represent many such systems with reasonably sized models. This section describes the model reduction techniques available in ASSIST. For a demonstration of these techniques and a discussion of their effectiveness, see reference 6.

13.1. Model Pruning

A model of a system with numerous components can have many long paths with one or two failures of each component type before a system-failure condition is reached. Because the occurrence of so many failures is unlikely during a short mission, these long paths typically contribute insignificant amounts to the probability of system failure. The dominant system-failure modes are typically the short paths to system failure that have failures of like components. Model pruning can be used to eliminate the long paths by conservatively assuming that system failure occurs earlier on those paths.

Model pruning is supported in ASSIST by the `PRUNEIF` statement, which is described in section 3.2.7. The pruning conditions are described by `PRUNEIF` statements much like the death conditions are described using `DEATHIF` statements. Like the death states, the pruned states are lumped according to which `PRUNEIF` statement was satisfied by the state.

Since the model is truncated by assuming system failure when the pruning criterion is met, this method is conservative. The system failure states caused by model pruning are lumped together according to the `PRUNEIF` statement that is satisfied; thus, the contribution to system-failure probability caused by pruning can be determined easily from the SURE output file.

13.2. Model Trimming

A new method for reducing the size of semi-Markov reliability models was recently developed by Dr. Allan White and Daniel Palumbo of NASA Langley Research Center. The details of this trimming method and the theorem proving that this method is conservative are given in reference 7. The user should read this paper to determine the system characteristics for which this trimming method is guaranteed to be conservative. Not all systems can be trimmed.

The `TRIM` statement is used in the setup or start section of the ASSIST input file to trim the model. The syntax is

TRIM OFF ;
or
TRIM ON [WITH <trimomega> **] ;**
or
TRIM FULL [WITH <trimomega> **] ;**

where <trimomega> is a real expression for the trimming bound Ω_{trim}.

The first form `TRIM OFF` is for no trimming and is the default.

The second form `TRIM ON` selects the trimming bound described in the White-Palumbo paper. In `TRIM ON` mode the model is altered so that each state in which the system is experiencing a recovery has all failure transitions from it that do not immediately end in a death state, ending in an intermediate trim state. There is a transition from the intermediate trim state to a final state at rate Ω_{trim} (<trimomega>). For example,

```
TRIM ON WITH 6e-4;
```

The `WITH` clause is optional; if it is absent, the user will be prompted for a value for Ω_{trim}. The value must be entered in response to the prompt and must be terminated with a semicolon.

The third form specifies use of the White-Palumbo trimming bound plus the generation of an extra transition from each pruned state to an additional pruned state at the same rate Ω_{trim}.

The user is again referred to reference 7 to justify trimming (TRIM ON or TRIM FULL) and to determine if this method is valid for the particular system class.

To be conservative, the <trimomega> value should be set as the sum of the failure rates of all system components. The justification for the TRIM FULL method is as follows: Each pruned state of the model is not a system death state. Thus, another failure of some component must occur before system failure can occur. The TRIM FULL feature simply includes this extra failure transition before pruning the model. To be conservative, <trimomega> must be greater than or equal to the sum of the failure rates for all components still in the system when any pruned state is reached.

13.3. Assignment of State Numbers

State numbers are assigned according to the following hierarchy:

1. When ONEDEATH is ON, death state numbers are always assigned first.
2. The pruned death states are assigned next.
3. If TRIM is ON or FULL (≥ 1), the TRIM death state is assigned next.
4. If TRIM is FULL (= 2), the individual pruned states are assigned next.
5. If TRIM is ON or FULL (≥ 1), the trim state is assigned next.
6. The start state is assigned next.

14. Model Generation Algorithm

The model generation algorithm builds the model from the start state by recursively applying the TRANTO rules. A list of states to be processed, called the *ready set*, begins with only the start state. Before application of a rule, ASSIST checks all the ASSERT conditions and prints any warning messages. All death conditions are then checked to determine if the current state is a death state. Since a death state denotes system failure, no transitions can leave a death state. If the state is not a death state, ASSIST then checks all pruned conditions to determine if the current state is a pruned state. If ASSIST finds a state-space variable that is out of range, or if ASSIST detects some other error in the state, the state is treated as a death state. Each TRANTO rule is then evaluated for the nondeath state. If the condition expression of the TRANTO rule evaluates as true for the current state, the destination expression is used to determine the state-space variable values of the destination state. If the destination state has not been defined in the model, the new state is added to the ready set of states to be processed. The transition rate is determined from the rate expression, and the transition description is printed to the model file. When all TRANTO rules have been applied to it, the state is removed from the ready set. When the ready set is empty, all possible paths terminate in death states and model building is complete.

By default, all death states are aggregated or lumped (ONEDEATH ON) according to the first DEATHIF statement to which the state conformed. If the user sets ONEDEATH OFF, then all distinct death states are kept in the model.

Note that the ready list is a subset of all state nodes that have been processed up to any given point. All state nodes that have been processed must remain in memory because ASSIST must check each new destination state to see if it has already been processed. There are typically many different paths to each model state. States are processed only once.

The following is a pseudo-code version of the algorithm used to generate the model:

```
(* ====================== subroutines/functions ==================== *)
FUNC  PROCESS(state,trim,fast,in_error)
      (* note that ``fast'' is ignored when trimming is off *)):
      state number ← search existing states.
      IF (state already present) THEN:
         is a death state if flagged as such.
      ELSE:
         save current state and dependent variable values.
            recompute dependent variables that are referenced in
                 ASSERT, DEATHIF, PRUNEIF sections for state.
            test all ASSERT's, printing WARNING message if a test fails.
            test all DEATHIF's.
            IF (not a death) test all PRUNEIF's.
         restore current state and dependent variable values.
         IF (death or prune state) AND (fast) AND (trimming is on) THEN:
            print warning message:
                  Model contains recovery transitions directly to death state
                  and therefore may not be suited to trimming.
         ENDIF.
      ENDIF.
      IF (death state) AND (lumping) THEN:
         state number ← death state number.
      ELSE IF (prune state) THEN:
         state number ← prune state number.
      ELSE IF ((trim) AND (trimming is on)) THEN:
         state number ← trim state number.
      ELSE (* not being lumped *):
         IF (death state) THEN:
            flag the state.
         ENDIF.
         IF (state does not yet exist) THEN:
            state number ← add the state to the ready list.
         ENDIF.
      ENDIF.
      RETURN state number.
ENDFUNC.

(* ======================== main algorithm ======================== *)

MAIN:
    (* process the start state *)
    compute start state.
%    compute dependent variables referenced in TRANTO section.
    start state number ← call PROCESS(start state,NORMAL,N/A,error).

    (* generate the model *)
    ready list ← pointer to first state.
    FOR current-state IN [all states on ready list] LOOP:
        IF (state is not flagged as death state) THEN:
           set fast transition counter to zero.
           FOR (all recovery (fast) TRANTO's) DO:
       compute new state.
              new state number ← call PROCESS(new state,NORMAL,FAST,error).
              print the transition to the model file.
              increment fast transition counter.
           ENDFOR.
```

```
            IF (trimming is on) THEN:
                set slow transition counter to zero.
                FOR (all non-recovery (slow) TRANTO's) DO:
                    compute new state.
                    IF (fast transition counter > 0) THEN:
                      new state number ← call PROCESS(new state,TRIM,SLOW,error).
                    ELSE:
                      new state number ← call PROCESS(new state,NORMAL,SLOW,
                                                      error).
                    ENDIF.
                    print the transition to the model file.
                ENDFOR.
            ELSE:
                set slow transition counter to zero.
                FOR (all non-recovery (slow) TRANTO's) DO:
                    compute new state.
                    new state number ← call PROCESS(new state,NORMAL,N/A).
                    print the transition to the model file.
                ENDFOR.
            ENDIF.
         ENDIF.
         print warning if no transitions out of a non-death state.
         ready list ← increment pointer to next ready state.
      ENDFOR.

      (* print extra trim transitions *)
      IF (trimming is on) THEN:
         FOR current-state IN [trim state only] DO:
             print transition from current-state to trim death
                   state BY TRIMOMEGA.
         ENDFOR.
         IF  (pruning with TRIMOMEGA (i.e., trim=2)) THEN
             FOR current-state IN [prune states] DO:
                 print transition from current-state to current prune
                       death state BY TRIMOMEGA.
             ENDFOR.
         ENDIF.
      ENDIF.
      FOR (all TRANTO's) DO:
         IF (never referenced) THEN:
             print a warning message that TRANTO was never used.
         ENDIF.
      ENDFOR.
      IF (fatal error occurred) THEN:
         flag model file as erroneous.
      ENDIF.
STOP.
```

15. File Processing

The ASSIST program reads an input file containing the model definition rules and creates several output files. The two most important output files are the model file and the log (listing) file.

The diagram in figure 8 illustrates the data information flow between files during the processing of an ASSIST input file. These files are described in sections 6.1 through 6.1.4.

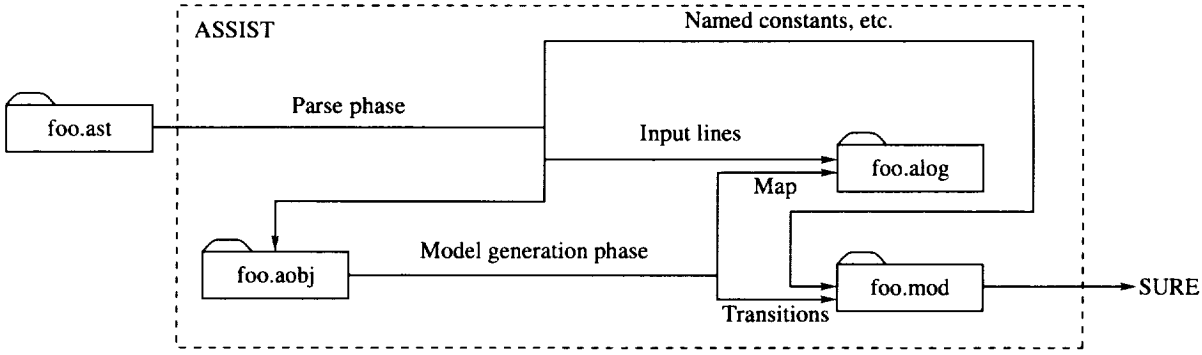


Figure 8. Data file flow in ASSIST.

15.1. Input File

The input file contains a sequence of ASSIST instructions, and the user must create the file by using a text editor or a tool such as TOTAL.[1]

By convention, the input filename must end with an **.ast** suffix. A file without this extension cannot be processed by ASSIST. When running ASSIST, the user may omit the suffix when typing in the file name. For example, either of the following will process the file **foo.ast**:

```
assist foo
assist foo.ast
```

The syntax and semantics of the ASSIST instructions are described below in detail.

15.1.1. Model File

The model output file produced by the program contains the semi-Markov model in the format needed for input to the SURE computer program. The model file has a **.mod** file extension.

The model file contains all the named numeric (non-Boolean) constants defined in the ASSIST input file setup and start sections as well as in the transitions generated by the ASSIST rule section. The ASSIST file layout and its three sections are discussed in the next chapter.

Any statements in the ASSIST input file that are surrounded by double quotation marks are also copied directly into the model file. These statements and named constants appear in the model file in the same sequence in which they appear in the ASSIST input file. Values input with the ASSIST `INPUT` statement are treated as named constants. All the transitions between states defining the semi-Markov model are listed next.

Each SURE state is defined by a unique integer. To make the model easier to understand, the model 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;
```

defines a transition from SURE state 1, which was ASSIST state (6,0,0), to SURE state 2, which was ASSIST state (5,1,0), at an exponential rate with mean λ. The comment feature can be turned off by adding the line `COMMENT=0;` to the ASSIST input file.

Following the state transitions are three comment lines containing the following statistics describing the model: the number of model states, the number of model transitions, and the number of distinct

[1]TOTAL is a prototype user interface program that generates ASSIST input files from a more abstract description of the system (ref. 8).

death states encountered in building the model. These statistics are printed within comment delimiters, and the SURE program will ignore them. Unless errors or warnings are encountered, these three comment lines will be the last three lines in the model file. For example,

```
(* NUMBER OF STATES IN MODEL = 2 *)
(* NUMBER OF TRANSITIONS IN MODEL = 6 *)
(* 6 DEATH STATES AGGREGATED INTO STATE 1 *)
```

If errors or warnings are detected, the number of errors and warnings are printed as comments at the end of the model file.

For example,

```
(* 0048 ERRORS *)
(* 0001 WARNING *)
```

15.1.2. Log File

The log (listing) file contains a listing of the ASSIST input file plus various information to aid the user in checking the correctness of the generated model. The listing file has the **.alog** file extension unless it is running on systems where four character extensions are illegal, in which case the listing file has the **.alg** file extension.

Each input line printed in the listing file is numbered for user convenience when the user is making corrections to a line. The user should make all corrections to the ASSIST input file (**.ast**) and not to the listing file.

The name of the ASSIST input file, followed by the date and time of execution, is printed in the upper left-hand corner of each page of the listing file. The page number is printed in the upper right-hand corner of each page of the listing file.

If any errors were encountered while parsing the ASSIST input file, they are printed after the input line containing the error, the model file is flagged as erroneous, and no transitions are generated. An attempt to run such a model file through SURE will be rejected. If any errors are encountered during model generation, the model file is flagged as erroneous. An attempt to run such a model file through SURE will be rejected. Errors encountered during model generation are printed after the input file listing and optional variable or load maps.

The following is a sample log file:

```
a3.ast    3/16/94 10:02:49 a.m.                    ASSIST 7.1, NASA LaRC    Page 1

(0001): SPACE = (A,B,C,D);
(0002): START = (2 OF 3,1 OF 2,1);
(0003): IF C = 2 TRANTO C = C+1 BY A+B+C+D;
(0004): DEATHIF C>2;
PARSING TIME = 0.15 sec.
RULE GENERATION TIME = 0.00 sec.
NUMBER OF STATES IN MODEL = 2
NUMBER OF TRANSITIONS IN MODEL = 1
1 DEATH STATES AGGREGATED INTO STATE 1
```

Note that the first line of each page begins with the name of the ASSIST input file (`a3.ast`), followed by the date and time of the run (`3/16/94 10:20:49 a.m.`), and ends with the program and version number which processed the input file (`ASSIST 7.1, NASA LaRC`), followed by the page number.

The input file is listed next. Note that the line numbers are not typed into the input file. They are automatically generated for reference purposes. The four corresponding lines in the input file `a3.ast` are

```
SPACE = (A,B,C,D);
START = (2 OF 3,1 OF 2,1);
IF C = 2 TRANTO C = C+1 BY A+B+C+D;
DEATHIF C>2;
```

The last five lines in the log file give statistics on the parsing time, rule generation time, number of model states, number of model transitions, and number of aggregated death states, respectively. Note that the number of aggregated death states is a count of how many transitions resulted in death, not a count of the number of unique state *n*-tuples which are death states.

Error messages appear in the log file directly below the line that is in error. The wedge (^) character points to the approximate location within the line where the error occurred. For example,

```
(0006): ABC = AAA - 2;
                ^ [ERROR] IDENTIFIER NOT DEFINED: AAA
(0007):
```

15.1.3. Object File

The object file contains binary numeric data tables and pseudo machine code. This file is currently used as a temporary file; however, future versions of ASSIST may make additional use of it.

The object file has the **.aobj** file extension unless it is running on systems where four-character extensions are illegal, in which case it has the **.aoj** file extension.

15.1.4. Temporary Files

The ASSIST program language uses several temporary files to process the ASSIST `INPUT` statements and to store variable definitions for the optional cross-reference map, which is listed on the log file. Temporary files are deleted automatically after successful generation of the model file. Temporary file names are system dependent. They begin with `QQ` and end with a sequence of digits. Temporary file names usually do not contain an extension.

16. Examples

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

16.1. Triad With Cold Spares

This sample architecture has a triad of processors in which all three processors execute the same program. In addition to the triad, the sample architecture contains a pool of two cold spare processors. Each processor receives identical inputs so that all nonfaulty processors produce the same output. The three outputs are voted and any incorrect outputs are masked by the voting as long as a majority of the active processors are nonfaulty. 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.

This architecture can be described in the ASSIST input language as follows:

```
(*  TRIAD WITH COLD SPARES  *)
N_PROCS = 3;              (* Number of active processors *)
N_SPARES = 2;             (* Number of 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   (* Replace failed processor with working spare *)
      TRANTO (NP, NFP-1, NS-1, NFS)
             BY FAST (1-(NFS/NS))*NFP*DELTA;
   IF NFS > 0    (* Replace failed processor with failed spare *)
      TRANTO (NP, NFP, NS-1, NFS-1)
             BY FAST (NFS/NS)*NFP*DELTA;
ENDIF;
```

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

- The failure rate of each active processor is λ_P.
- The failure rate of a cold spare processor is λ_S.
- A failed active processor is replaced by a spare processor at fast exponential rate δ.

The third transition type 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:

- System failure occurs unless a majority of the active processors are nonfaulty.

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

The following dialog, consisting of an execution of ASSIST followed by an execution of SURE will process the input file:

```
% assist triadcold
ASSIST VERSION 7.1                                  NASA Langley Research Center
PARSING TIME = 0.18 sec.
generating SURE model file...
RULE GENERATION TIME = 0.02 sec.
NUMBER OF STATES IN MODEL = 13
NUMBER OF TRANSITIONS IN MODEL = 24
6 DEATH STATES AGGREGATED INTO STATE 1

% sure

  SURE V7.9.8   NASA Langley Research Center

  1? read0 triadcold;

       0.03 SECS. TO READ MODEL FILE
 37? run;
 MODEL FILE = triadcold.mod                SURE V7.9.8 14 Sep 93  17:11:45
```

```
                LOWERBOUND     UPPERBOUND      COMMENTS                    RUN #1
 -----------    -----------    -----------     ---------------------------------
                1.66645e-10    1.69427e-10     <prune 1.2e-19>

 25 PATH(S) TO DEATH STATES 2 PATH(S) PRUNED
 HIGHEST PRUNE LEVEL =  2.03357e-17
 0.000 SECS. CPU TIME UTILIZED
 38? exit;
```

The semi-Markov model is shown in figures 9 and 10.

The log file that is generated from executing ASSIST on this input file is

```
triadcold.ast    3/21/94 2:44:32 p.m.          ASSIST 7.1, NASA LaRC        Page 1

(0001): (*  TRIAD WITH COLD SPARES  *)
(0002):
(0003): N_PROCS = 3;              (* Number of active processors *)
(0004): N_SPARES = 2;             (* Number of spare processors *)
(0005): LAMBDA_P = 1E-4;          (* Failure rate of active processors *)
(0006): LAMBDA_S = 1E-5;          (* Failure rate of spare processors *)
(0007): DELTA = 3.6E3;            (* Reconfiguration rate *)
(0008):
(0009): SPACE = (NP: 0..N_PROCS,          (* Number of active processors *)
(0010):          NFP: 0..N_PROCS,         (* Number of failed active processors *)
(0011):          NS: 0..N_SPARES,         (* Number of spare processors *)
(0012):          NFS: 0..N_SPARES);       (* Number of failed spare processors *)
(0013):
(0014): START = (N_PROCS, 0, N_SPARES, 0);
(0015):
(0016): DEATHIF 2 * NFP >= NP;
(0017):
(0018): IF NP > NFP TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_P;
(0019):    (* Active processor failure *)
(0020):
(0021): IF NS > NFS TRANTO NFS = NFS+1 BY NS*LAMBDA_S;
(0022):    (* Spare processor failure *)
(0023):
(0024): IF (NFP > 0 AND NS > 0) THEN
(0025):    IF NS > NFS    (* Replace failed processor with working spare *)
(0026):       TRANTO (NP, NFP-1, NS-1, NFS)
(0027):              BY FAST (1-(NFS/NS))*NFP*DELTA;
(0028):    IF NFS > 0     (* Replace failed processor with failed spare *)
(0029):       TRANTO (NP, NFP, NS-1, NFS-1)
(0030):              BY FAST (NFS/NS)*NFP*DELTA;
(0031): ENDIF;

PARSING TIME = 0.21 sec.
RULE GENERATION TIME = 0.02 sec.
NUMBER OF STATES IN MODEL = 13
NUMBER OF TRANSITIONS IN MODEL = 24
6 DEATH STATES AGGREGATED INTO STATE 1
```

The model file generated by ASSIST for this example is

```
N_PROCS = 3;
N_SPARES = 2;
LAMBDA_P = 1E-4;
LAMBDA_S = 1E-5;
DELTA = 3.6E3;
```

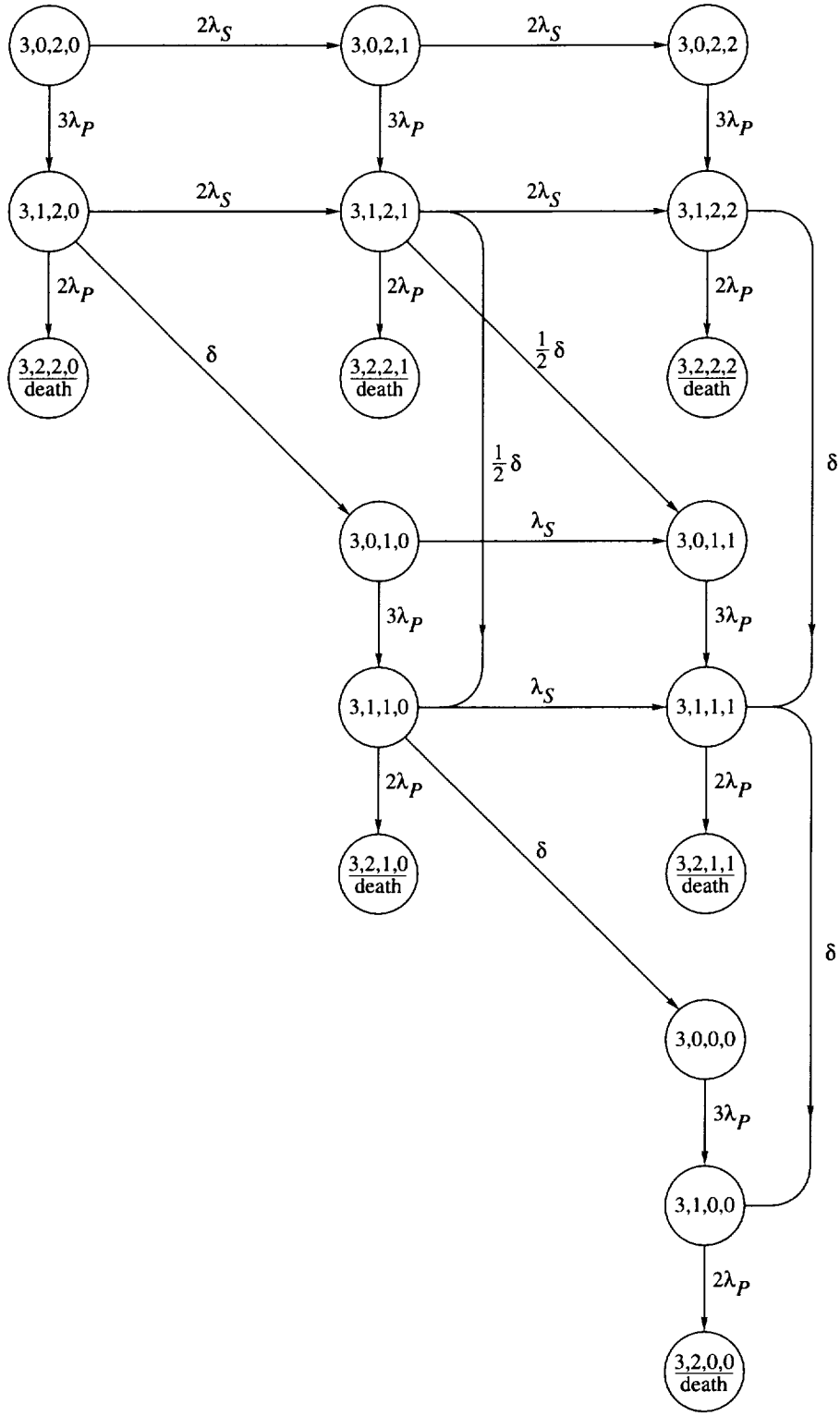


Figure 9. Semi-Markov triad model with cold spares (ASSIST state numbers).

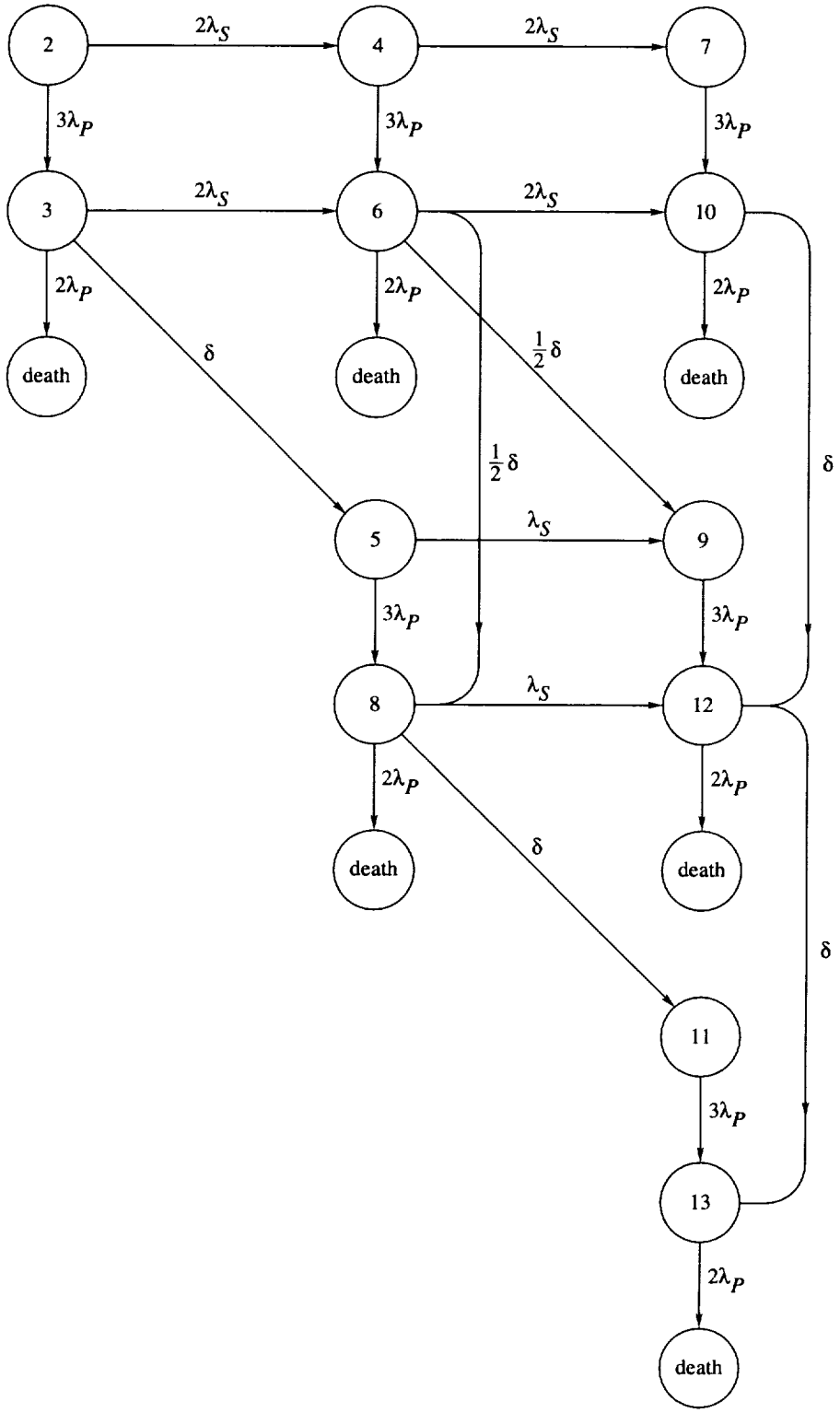


Figure 10. Semi-Markov triad model with cold spares (SURE state numbers).

```
    2(* 3,0,2,0 *),     3(* 3,1,2,0 *)          = (3-0)*LAMBDA_P;
    2(* 3,0,2,0 *),     4(* 3,0,2,1 *)          = 2*LAMBDA_S;
    3(* 3,1,2,0 *),     5(* 3,0,1,0 *)          = FAST (1-(0/2))*1*DELTA;
    3(* 3,1,2,0 *),     1(* 3,2,2,0 DEATH  *)  = (3-1)*LAMBDA_P;
    3(* 3,1,2,0 *),     6(* 3,1,2,1 *)          = 2*LAMBDA_S;
    4(* 3,0,2,1 *),     6(* 3,1,2,1 *)          = (3-0)*LAMBDA_P;
    4(* 3,0,2,1 *),     7(* 3,0,2,2 *)          = 2*LAMBDA_S;
    5(* 3,0,1,0 *),     8(* 3,1,1,0 *)          = (3-0)*LAMBDA_P;
    5(* 3,0,1,0 *),     9(* 3,0,1,1 *)          = 1*LAMBDA_S;
    6(* 3,1,2,1 *),     9(* 3,0,1,1 *)          = FAST (1-(1/2))*1*DELTA;
    6(* 3,1,2,1 *),     8(* 3,1,1,0 *)          = FAST (1/2)*1*DELTA;
    6(* 3,1,2,1 *),     1(* 3,2,2,1 DEATH  *)  = (3-1)*LAMBDA_P;
    6(* 3,1,2,1 *),    10(* 3,1,2,2 *)          = 2*LAMBDA_S;
    7(* 3,0,2,2 *),    10(* 3,1,2,2 *)          = (3-0)*LAMBDA_P;
    8(* 3,1,1,0 *),    11(* 3,0,0,0 *)          = FAST (1-(0/1))*1*DELTA;
    8(* 3,1,1,0 *),     1(* 3,2,1,0 DEATH  *)  = (3-1)*LAMBDA_P;
    8(* 3,1,1,0 *),    12(* 3,1,1,1 *)          = 1*LAMBDA_S;
    9(* 3,0,1,1 *),    12(* 3,1,1,1 *)          = (3-0)*LAMBDA_P;
   10(* 3,1,2,2 *),    12(* 3,1,1,1 *)          = FAST (2/2)*1*DELTA;
   10(* 3,1,2,2 *),     1(* 3,2,2,2 DEATH  *)  = (3-1)*LAMBDA_P;
   11(* 3,0,0,0 *),    13(* 3,1,0,0 *)          = (3-0)*LAMBDA_P;
   12(* 3,1,1,1 *),    13(* 3,1,0,0 *)          = FAST (1/1)*1*DELTA;
   12(* 3,1,1,1 *),     1(* 3,2,1,1 DEATH  *)  = (3-1)*LAMBDA_P;
   13(* 3,1,0,0 *),     1(* 3,2,0,0 DEATH  *)  = (3-1)*LAMBDA_P;

(* NUMBER OF STATES IN MODEL = 13 *)
(* NUMBER OF TRANSITIONS IN MODEL = 24 *)
(* 6 DEATH STATES AGGREGATED INTO STATE 1 *)
```

16.2. Many Triads With 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, 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, 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 constructs so that all triads are handled. The only significant model changes 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.

```
(*  MULTIPLE TRIADS WITH POOL OF SPARES *)

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

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..N_PROCS, (* processors each triad *)
         NFP: ARRAY[1..N_TRIADS] OF 0..N_PROCS, (* active each 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 failure *)

FOR J IN [1..N_TRIADS];
   IF NP[J] > NFP[J] TRANTO NFP[J]++
        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]--,NS--
            BY FAST (1-(NFS/NS))*NFP[J]*DELTA;
            (* Replace failed processor with working spare *)

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

      ELSE
         TRANTO NP[J]=0,NFP[J]=0,NS=NP[J]-NFP[J],NT-- BY FAST OMEGA;
            (* 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 shown in the example above, the number of initial spares is set with the constant N_SPARES. Table 10 shows that changing these two constants has a dramatic effect on the number of states in the generated model.

Table 10. Number of Model States for Various Initial Configurations (With ONEDEATH OFF)

Number of triads	Number of model states for spares numbering—					
	0	1	2	3	4	5
1	4	10	19	31	46	64
2	45	61	85	117	157	205
3	219	259	319	399	499	619
4	889	985	1129	1321	1561	1849
5	3323	3547	3883	4331	4891	5563

16.3. Quad With Transient Faults

A quad architecture with both permanent and transient faults is modeled in this example. The system behavior is as follows:

1. Permanent faults arrive at rate λ_P.
2. Transient faults arrive at rate λ_T.
3. Transient faults disappear at fast exponential rate δ_D.

4. Reconfiguration of processors with permanent faults or transient faults that remain long enough to be detected occurs at fast exponential rate δ_R.

5. System failure occurs when a majority of the processors have permanent or transient faults.

The ASSIST input file that describes this system is as follows:

```
(* QUAD WITH TRANSIENT FAULTS *)

NP = 4;                  (* Number of processors *)
LAMBDA_P = 1E-4;         (* Permanent fault arrival rate *)
LAMBDA_T = 10*LAMBDA_P; (* Transient fault arrival rate *)
DELTA_D = .5;            (* Transient fault disappearance rate *)
DELTA_R = 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);

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

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

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

IF NFP > 0 TRANTO NFP-- BY FAST DELTA_R;(* Permanent fault reconfiguration *)
```

The model that is generated for this example contains 15 states and 20 transitions. This ASSIST input 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.

16.4. Monitored Sensor Failure

A triad of monitored sensors is modeled in this example. Faulty sensors are detected by majority voting. If half the active sensors are faulty and the vote could produce a tie, the monitors are used to detect which sensor is faulty. Detection using the monitors has imperfect coverage. The system behavior is as follows:

1. Sensors fail at rate λ_S.

2. Monitors fail at rate λ_M.

3. If a majority of the sensors are nonfaulty, a failed sensor is removed with a mean time of μ_1 and a standard deviation of σ_1. When majority voting is used for fault detection, the probability of detecting the correct sensor as faulty is 1.

4. If exactly half the sensors are faulty, a sensor is removed with a mean time of μ_2 and a standard deviation of σ_2. In this case the probability that the nonfaulty sensor was the one removed is 0.98.

5. The system fails if the majority of sensors fail, or if half the sensors fail and the number of monitors is less than the number of sensors.

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

```
(* MONITORED SENSOR FAILURE MODEL *)

LAMBDA_S = 10E-6; (* Failure rate of sensors *)
LAMBDA_M = 1E-6;  (* Failure rate of monitors *)
MU_1 = 3E-4;      (* Mean recovery time for first fault *)
SIG_1 = 1E-4;     (* S.D. of recovery time for first fault *)
MU_2 = 1E-4;      (* Mean recovery time for second fault *)
SIG_2 = 2E-5;     (* S.D. of recovery time for second fault *)
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 2*NFS > NS; (* Majority of sensors failed *)
DEATHIF 2*NFS = NS AND NM < NS;
   (* Half of the sensors fail and the number of monitors *)
   (*   working is less than the number of active sensors *)

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

IF NS>2*NFS AND NFS>0 THEN   (* First fault recovery *)
   IF NM>0 TRANTO NS--,NFS--,NM-- BY <MU_1,SIG_1,(NM/NS)>;
           (* Loss of monitored sensor *)
   IF NM<NS TRANTO NS--,NFS-- BY <MU_1,SIG_1,(NS-NM)/NS>;
           (* Loss of unmonitored sensor *)
ENDIF;

IF NS=2*NFS AND NFS>0 THEN   (* Second fault recovery *)
   TRANTO NS--,NFS-- BY <MU_2,SIG_2,COV_2>;
          (* Successfully removed failed sensor *)
   TRANTO NS-- BY <MU_2,SIG_2,1.0-COV_2>;
          (* Mistakenly removed nonfaulty sensor *)
ENDIF;
```

The semi-Markov model that is generated for this example contains 20 states and 26 transitions.

16.5. Two Triads With Three Power Supplies

This example has two computer triads and one triad of power supplies that are connected so that one computer in each triad is connected to each power supply. Thus, if a power supply fails, one computer in each triad fails. Because of the complex failure dependencies, this is a difficult 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 method 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:

```
(* 2 TRIADS OF COMPUTERS WITH 1 TRIAD OF POWER SUPPLIES        *)
(* CONNECTED SUCH THAT 1 COMPUTER IN EACH TRIAD IS CONNECTED TO  *)
(* EACH POWER SUPPLY.  THUS, IF A POWER SUPPLY FAILS, THEN ONE *)
(* COMPUTER IN EACH TRIAD FAILS.  THE SYSTEM FAILS IF EITHER   *)
(* TRIAD OF COMPUTERS FAILS.                                   *)
```

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

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

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

FOR I IN [1..3]
   IF (NOT CAF[I])
      TRANTO CAF[I]=TRUE BY LAM_C;  (* Computer failure in Triad A *)
   IF (NOT CBF[I])
      TRANTO CBF[I]=TRUE BY LAM_C;  (* Computer failure in Triad B *)
   IF (NOT PSF[I])
      TRANTO CAF[I]=TRUE,CBF[I]=TRUE,PSF[I]=TRUE
             BY LAM_PS;  (* Power supply failure *)
ENDFOR;
```

This 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 systems with semi-Markov models is still an art, even when using the ASSIST program. As with any language, once the user becomes proficient in using the ASSIST language, he can see more easily 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 practice sometimes leads to rather complex TRANTO and DEATHIF statements. By using state-space variables to represent the number of failed computers in each triad and by adding a flag to signal the dependencies between failed computers, the system may be modeled with a much smaller state space. When the user combines the resulting complex transition rules by using logical reasoning, the system described above can be modeled by the following input file:

```
(* 2 TRIADS OF COMPUTERS WITH 1 TRIAD OF POWER SUPPLIES        *)
(* CONNECTED SO THAT 1 COMPUTER IN EACH TRIAD IS CONNECTED TO  *)
(* EACH POWER SUPPLY.  THUS, IF A POWER SUPPLY FAILS, THEN ONE *)
(* COMPUTER IN EACH TRIAD FAILS.  THE SYSTEM FAILS IF EITHER   *)
(* TRIAD OF COMPUTERS FAILS.                                   *)

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, (* Count of failed computers each triad *)
         NFS: 0..3,                (* Count of failed power supplies *)
         SAME:BOOLEAN);            (* Set if all failed computers fall
                                      on same power supply *)

FUNCTION OTHER(II) = 3-II;

START = (2 OF 0,
         0,
         TRUE);

FOR IX IN [1..2]
   DEATHIF NFP[IX]>1;
   IF (NFP[OTHER(IX)] > 0) THEN  (* Other triad has a failed computer *)
      TRANTO NFP[IX]++ BY LAM_C;
         (* Failure of computer on same power supply as other failed one *)
      TRANTO NFP[IX]++,SAME=FALSE BY (2-NFP[IX])*LAM_C;
         (* Failures of computers on different power *)
         (* supplies than the other failed one       *)
```

```
   ELSE
      TRANTO NFP[IX]++ BY (3-NFP[IX])*LAM_C;
         (* Failures of computers when other triad has no failures yet *)
   ENDIF;
ENDFOR;

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

The second ASSIST input file leads to a semi-Markov model with only 25 states and 29 transitions, as compared to the first strategy, which required 70 states and 138 transitions to model the same system. However, this input file is much more difficult to understand and verify. Automatic lumping of death states further reduces the model to 7 states with 29 transitions.

16.6. Triad With Intermittent Faults

The next example shows a triad subject to intermittent faults. This triad has three processors with intermittent fault arrival rates of λ_I. When an intermittent fault arrives, it is assumed to be in an active state; i.e., it is actively producing errors. After arrival, intermittent faults change to a benign state at fast exponential rate β and change to an active state again at fast exponential rate α.

A solution using STEM or PAWS is necessary because the SURE program cannot handle the fast loops in this model.

```
                        (*************************)
                        (*************************)
                        (***                   ***)
                        (***   RATE  CONSTANTS   ***)
                        (***                   ***)
                        (*************************)
                        (*************************)

L_I_triad    = 0.707e-4; (* Arrival rate of intermittent fault for "triad" *)
ALP_triad    = 2.718281e3; (* Rate benign "triad" intermittent goes active *)
BET_triad    = 3.141592e3; (* Rate active "triad" intermittent goes benign *)

                        (*************************)
                        (*************************)
                        (***                   ***)
                        (***   SPACE CONSTANTS   ***)
                        (***                   ***)
                        (*************************)
                        (*************************)
```

```
NI_triad     = 3;    (* Redundancy count for "triad" *)
NTOT = NI_triad;     (* Total number of components initially *)

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   STATE SPACE DEFINED   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

SPACE =
  (
   NW_triad     : 0..NI_triad, (* Count of working in "triad" *)
   NFA_triad    : 0..NI_triad, (* Count of active failed in "triad" *)
   NFB_triad    : 0..NI_triad  (* Count of benign failed in "triad" *)
  );

IMPLICIT TFA_triad[NFA_triad] = (* Total active failed in "triad" *)
         NFA_triad;
IMPLICIT TWA_triad[NW_triad,NFB_triad] = (* Total active working in "triad" *)
         NW_triad + NFB_triad;
IMPLICIT TA_triad[NW_triad,NFB_triad,NFA_triad] = (* Total active in "triad" *)
         TFA_triad + TWA_triad;

START =
  (
   NI_triad,       (* Count of working in "triad" *)
   0,              (* Count of active failed in "triad" *)
   0               (* Count of benign failed in "triad" *)
  );

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   COMPONENT:  "triad"   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

ASSERT (TWA_triad + TFA_triad) = NI_triad;

IF (NW_triad > 0)
   TRANTO NW_triad--,NFA_triad++
          BY NW_triad*L_I_triad;  (* Intermittent fault arrival *)

IF (NFB_triad > 0)
   TRANTO NFB_triad--,NFA_triad++
          BY FAST ALP_triad;   (* Benign fault goes active *)

IF (NFA_triad > 0) THEN  (* Active Intermittent faults *)
   TRANTO NFA_triad--,NFB_triad++
          BY FAST BET_triad;   (* Active fault goes benign *)
ENDIF;

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   MISC. DEATH & PRUNE   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

DEATHIF TFA_triad >= TWA_triad;
```

16.7. Degradable Triad With Intermittent Faults

The next example is a degradable triad subject to intermittent faults. This triad has three processors with intermittent fault arrival rates of λ_I. When an intermittent fault arrives, it is assumed to be in an active state; i.e., it is actively producing errors. After arrival, intermittent faults change to a benign state at fast exponential rate β and change to an active state again at fast exponential rate α. Active processors that are faulty are removed from the system at fast exponential rate ρ_D.

A solution using STEM or PAWS is necessary because the SURE program cannot handle the fast loops in this model.

```
                        (*************************)
                        (*************************)
                        (***                   ***)
                        (***   RATE  CONSTANTS   ***)
                        (***                   ***)
                        (*************************)
                        (*************************)
R_D_triad     = 1.414e3; (* Rate to degrade for "triad" *)
L_I_triad     = 0.707e4; (* Arrival rate of intermittent fault for "triad" *)
ALP_triad     = 2.718281e3; (* Rate benign "triad" intermittent goes active *)
BET_triad     = 3.141592e3; (* Rate active "triad" intermittent goes benign *)
                        (*************************)
                        (*************************)
                        (***                   ***)
                        (***   SPACE CONSTANTS   ***)
                        (***                   ***)
                        (*************************)
                        (*************************)
NI_triad      = 3;    (* Redundancy count for "triad" *)
NTOT = NI_triad;      (* Total number of components initially *)
                     (*******************************)
                     (*******************************)
                     (***                         ***)
                     (***   STATE SPACE DEFINED   ***)
                     (***                         ***)
                     (*******************************)
                     (*******************************)
SPACE =
  (
   NW_triad      : 0..NI_triad, (* Count of working in "triad" *)
   NFA_triad     : 0..NI_triad, (* Count of active failed in "triad" *)
   NFB_triad     : 0..NI_triad  (* Count of benign failed in "triad" *)
  );
IMPLICIT TFA_triad[NFA_triad] = (* Total active failed in "triad" *)
         NFA_triad;
IMPLICIT TWA_triad[NW_triad,NFB_triad] = (* Total active working in "triad" *)
         NW_triad + NFB_triad;
IMPLICIT TA_triad[NW_triad,NFB_triad,NFA_triad] = (* Total active in "triad" *)
         TFA_triad + TWA_triad;
START =
  (
   NI_triad,        (* Count of working in "triad" *)
   0,               (* Count of active failed in "triad" *)
   0                (* Count of benign failed in "triad" *)
  );
```

```
                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   COMPONENT:  "triad"   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

ASSERT (TWA_triad + TFA_triad) <= NI_triad;

IF (NW_triad > 0)
   TRANTO NW_triad--,NFA_triad++
          BY NW_triad*L_I_triad;  (* Intermittent fault arrival *)

IF (NFB_triad > 0)
   TRANTO NFB_triad--,NFA_triad++
          BY FAST ALP_triad;   (* Benign fault goes active *)

IF (NFA_triad > 0) THEN  (* Active Intermittent faults *)
   TRANTO NFA_triad--,NFB_triad++
          BY FAST BET_triad;   (* Active fault goes benign *)
   IF ((NW_triad + NFB_triad) = 2) THEN (* degrade to simplex *)
      IF (NW_triad = 2) THEN
         TRANTO NFA_triad--,NW_triad=1 BY FAST R_D_triad;
      ELSE IF (NW_triad = 0) THEN
         TRANTO NFA_triad--,NFB_triad=1 BY FAST R_D_triad;
      ELSE
         TRANTO NFA_triad--,NW_triad=0 BY FAST 0.5*R_D_triad;
         TRANTO NFA_triad--,NFB_triad=0 BY FAST 0.5*R_D_triad;
      ENDIF; ENDIF;
   ELSE (* degrade by one *)
      TRANTO NFA_triad-- BY FAST R_D_triad;
   ENDIF;
ENDIF;

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   MISC. DEATH & PRUNE   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

DEATHIF TFA_triad >= TWA_triad;
```

16.8. Triad With Hot/Warm/Cold Spares

The next example is a triad with hot, warm, and cold spares. The three processors in the triad have permanent fault arrival rates of λ_P. If available, hot spares are taken first, warm next, and cold last. Faulty processors are replaced with spares at fast exponential rate δ_{hot}, δ_{warm}, or δ_{cold}. Failure rates for hot and warm spares are given by λ_{hot} and λ_{warm}, respectively. Cold spares are optimistically assumed to have failure rates of zero.

```
                        (*************************)
                        (*************************)
                        (***                   ***)
                        (***   RATE  CONSTANTS ***)
                        (***                   ***)
                        (*************************)
                        (*************************)
```

```
L_P_triad     = 1.e-4;   (* Arrival rate of permanent fault for "triad" *)
DEL_hot       = 7.011e6; (* Rate to reconfigure "hot" spare into "triad" *)
DEL_warm      = 6.802e6; (* Rate to reconfigure "warm" spare into "triad" *)
DEL_cold      = 2.114e6; (* Rate to reconfigure "cold" spare into "triad" *)

LAM_hot       = 1.2e-4;  (* Arrival rate of permanent "hot" spare faults *)
LAM_warm      = 1.0e-8;  (* Arrival rate of permanent "warm" spare faults *)

                         (*****************************)
                         (*****************************)
                         (***                       ***)
                         (***   SPACE CONSTANTS     ***)
                         (***                       ***)
                         (*****************************)
                         (*****************************)

NI_triad      = 3;    (* Redundancy count for "triad" *)
NSI_hot       = 2;    (* Initial pool size for "hot" *)
NSI_warm      = 5;    (* Initial pool size for "warm" *)
NSI_cold      = 10;   (* Initial pool size for "cold" *)
                      (* Total number of components initially *)

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   STATE SPACE DEFINED   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

SPACE =
  (
   NW_triad     : 0..NI_triad,  (* Count of working in "triad" *)
   NWS_hot      : 0..NSI_hot,   (* Working spares count, HOT pool "hot" *)
   NFS_hot      : 0..NSI_hot,   (* Failed spares count, HOT pool "hot" *)
   NWS_warm     : 0..NSI_warm,  (* Working spares count, WARM pool "warm" *)
   NFS_warm     : 0..NSI_warm,  (* Failed spares count, WARM pool "warm" *)
   NS_cold      : 0..NSI_cold   (* Spares count, COLD pool "cold" *)
  );

IMPLICIT TFA_triad[NW_triad] = (* Total active failed in "triad" *)
         NI_triad - NW_triad;
IMPLICIT TWA_triad[NW_triad] = (* Total active working in "triad" *)
         NW_triad;
IMPLICIT TA_triad[] = NI_triad;
IMPLICIT NS_triad[NWS_hot,NFS_hot,NWS_warm,NFS_warm,NS_cold] =
         (* Total spares available to "triad" *)
         NWS_hot + NFS_hot + NWS_warm + NFS_warm + NS_cold;

IMPLICIT NS_hot[NWS_hot,NFS_hot] = NWS_hot + NFS_hot;
IMPLICIT PRW_hot[NWS_hot,NFS_hot] = NWS_hot / NS_hot;
IMPLICIT PRF_hot[NWS_hot,NFS_hot] = NFS_hot / NS_hot;

IMPLICIT NS_warm[NWS_warm,NFS_warm] = NWS_warm + NFS_warm;
IMPLICIT PRW_warm[NWS_warm,NFS_warm] = NWS_warm / NS_warm;
IMPLICIT PRF_warm[NWS_warm,NFS_warm] = NFS_warm / NS_warm;

START =
  (
   NI_triad,       (* Count of working in "triad" *)
   NSI_hot,        (* Working spares count, HOT pool "hot" *)
```

```
   0,              (* Failed spares count, HOT pool "hot" *)
   NSI_warm,       (* Working spares count, WARM pool "warm" *)
   0,              (* Failed spares count, WARM pool "warm" *)
   NSI_cold        (* Spares count, COLD pool "cold" *)
  );

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   COMPONENT:  "triad"   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

IF (NW_triad > 0)
   TRANTO NW_triad--
          BY NW_triad*L_P_triad;  (* Permanent fault arrival *)

IF (TFA_triad > 0) THEN  (* Faults present *)
   IF (NS_hot > 0) THEN   (* Try primary spare pool "hot" first *)
      IF (NWS_hot > 0)
         TRANTO NWS_hot--,NW_triad++
                BY FAST PRW_hot*DEL_hot;
      IF (NFS_hot > 0)
         TRANTO NFS_hot--
                BY FAST PRF_hot*DEL_hot;
   ELSE IF (NS_warm > 0) THEN   (* Try secondary spare pool "warm" next *)
      IF (NWS_warm > 0)
         TRANTO NWS_warm--,NW_triad++
                BY FAST PRW_warm*DEL_warm;
      IF (NFS_warm > 0)
         TRANTO NFS_warm--
                BY FAST PRF_warm*DEL_warm;
   ELSE IF (NS_cold > 0) THEN   (* Try tertiary spare pool "cold" last *)
      TRANTO NS_cold--,NW_triad++ BY FAST DEL_cold;
   ENDIF; ENDIF; ENDIF;
ENDIF;

                         (*****************************)
                         (*****************************)
                         (***                       ***)
                         (***   SPARE POOL: "hot"   ***)
                         (***                       ***)
                         (*****************************)
                         (*****************************)

ASSERT (NWS_hot + NFS_hot) <= NSI_hot;

IF (NWS_hot > 0)   (* Arrival of permanent "hot" spare fault *)
   TRANTO NWS_hot--,NFS_hot++ BY NWS_hot*LAM_hot;

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   SPARE POOL:  "warm"   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)
```

```
ASSERT (NWS_warm + NFS_warm) <= NSI_warm;

IF (NWS_warm > 0)    (* Arrival of permanent "warm" spare fault *)
   TRANTO NWS_warm--,NFS_warm++ BY NWS_warm*LAM_warm;

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   SPARE POOL:  "cold"   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

ASSERT NS_cold <= NSI_cold;

                        (*******************************)
                        (*******************************)
                        (***                         ***)
                        (***   MISC. DEATH & PRUNE   ***)
                        (***                         ***)
                        (*******************************)
                        (*******************************)

DEATHIF TFA_triad >= TWA_triad;
```

17. Command-Line Parameters and Options

This section details the command for executing the ASSIST program. The program is written in the C programming language and will execute on a VAX under the VMS operating system or on a Sun-3 or Sun SPARCstation[2] under the UNIX operating system. An ANSI-standard C compiler is required to compile the program. The current Sun C compiler will not compile ASSIST, but Sun is planning to release an ANSI C compiler in the future. However, an ANSI C compiler for the Sun computers **gcc**, is available from the Free Software Foundation. The C compiler available with the VAX/VMS operating system is an ANSI C compiler.

When running the ASSIST program, the only positional parameter is the name of the input file being processed. The **.ast** suffix may be omitted from the file name on the ASSIST command line. Options are also allowed. The file name may not follow an option but must precede all options.

Options must be preceded by a slash under VMS, as in

/map

and must be preceded by a dash under UNIX, as in

-map

Options may be specified either in upper or lower case. The normal UNIX case sensitivity is not enforced on the ASSIST command-line options.

The syntax is

assist
or
assist <filename>
or
assist <option-list>
or
assist <filename> <option-list>

[2]SPARCstation: a trademark of SPARC International.

Respective examples are

assist
or
assist foo
or
assist –xref
or
assist foo –xref

If the file name is omitted, ASSIST will prompt for one.

There are several options that may be listed in any order on the command line. These options are all enumerated in appendix E. A few important options are discussed below.

17.1. Controlling Error/Warning Limits

The **–lel** option controls the maximum number of errors allowed on a single ASSIST input line. The **–el** option controls the maximum cumulative number of errors allowed throughout the whole input file. The **–lwl** option controls the maximum number of warnings allowed on a single ASSIST input line. The **–wl** option controls the maximum cumulative number of warnings allowed throughout the entire input file.

Examples are

```
assist foo –el=80
assist foo –el=100 –wl=1000 –lel=10 –lwl=10
```

Defaults are

```
assist foo –el=40 –wl=40 –lel=5 –lwl=5
```

17.2. Changing Allowable Input-Line Width

The **–wid** option changes the allowable length of an input line. The default width is 79. The number 79 was chosen because input lines are echoed to the terminal when an error occurs and because some terminals wrap/truncate with column 80. The value can be changed as illustrated in the example:

```
assist foo –wid=133
```

17.3. Generating Identifier Cross-Reference Map

The **–xref** (synonymous with **–map**) option causes inclusion of an identifier cross-reference map in the log (**.log**) file. The map follows the input listing and includes the line declared in, line(s) defined (given a value) in, and lines referenced in. Also referenced are matching ELSE and ENDIF and matching ENDFOR key words.

In a cross-reference listing, DCL stands for *declared* and indicates the line on which the item was declared, SET indicates the line(s) on which the item took on a value, and USE indicates the line(s) on which the item was referenced (*used*).

The following shows the syntax of input-line references:

<line-number> **c** <column-number>

The following log file was produced with **–xref**:

```
sys1.ast    3/17/94 9:46:50 a.m.                ASSIST 7.1, NASA LaRC    Page 1

(0001): LAMBDA = 1.e-4;
(0002): NP = 3;
(0003):
```

```
(0004): SPACE = (NW:0..NP);
(0005): START = (NP);
(0006):
(0007): IF (NW > 0) THEN
(0008):    TRANTO NW-- BY NW*LAMBDA;
(0009): ENDIF;
(0010):
(0011): DEATHIF (NP - NW) >= NW;

PARSING TIME = 0.14 sec.
RULE GENERATION TIME = 0.00 sec.
NUMBER OF STATES IN MODEL = 3
NUMBER OF TRANSITIONS IN MODEL = 2
1 DEATH STATES AGGREGATED INTO STATE 1
```

17.4. Piping Model to Standard Output

The **–pipe** option is used on UNIX systems to write the model to standard output so that it can be piped directly to SURE. For example, to process the input file **foo.ast** and pipe the model directly to SURE for solution, use the command

```
assist foo –pipe | sure > sureout
```

When the **–pipe** option is specified, dialog which normally appears on the standard output is redirected to the standard error.

The **–nostats** option can be used together with the **–pipe** option to suppress the printing of ASSIST statistics. This is useful when the output from SURE is to come directly to standard output and would otherwise be scrambled with the standard error file statistics from ASSIST. For example

```
assist foo –pipe –nostats | sure
```

The model output from ASSIST should not be piped directly to SURE unless all errors and warnings have been fixed. If this is done, the standard output from SURE will be scrambled with the standard error messages from ASSIST. The **–nostats** option will not suppress error and warning messages.

If the VMS (or other non-UNIX) user attempts to use the **/pipe** option, a warning message will be printed:

```
[WARNING] OPTION ONLY VALID ON A UNIX SYSTEM.  OPTION IGNORED: /pipe
```

17.5. Batch Mode

The **–bat** option causes ASSIST to run in batch mode. In batch mode, the ASSIST command line is echoed to standard error. A batch process will usually redirect standard error to a file instead of to the user's monitor screen. If the batch process invokes ASSIST many times, and one of the input files happens to contain some errors, the user will be able to tell which file was being processed by examining the echoed command line.

17.6. Controlling Printing of Warning Messages

The **–w** option specifies the number of warning levels that are printed. The lower the number of warning levels reported, the fewer warning messages there are.

18. Concluding Remarks

The ASSIST program automatically generates a semi-Markov reliability model from an abstract input language. Semi-Markov models have the flexibility to accurately represent virtually any fault-tolerant system. The ASSIST input language provides an abstraction for describing rules rather than

individual transitions. However, the flexibility of semi-Markov modeling is maintained because no assumptions are made about the system; the rules themselves completely specify the model to be generated.

As flight-critical systems become more complex and more highly integrated, the Markov models describing them will become enormously complex. However, our experience has been that even the most complex system characteristics can usually 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. Furthermore, the process of describing a system in terms of the model generation rules forces the reliability engineer to clearly understand the fault-tolerance strategies of the system, and the abstract description is also useful for communicating and validating the system model.

Further levels of abstraction are both possible and feasible. Several research prototype computer programs to generate ASSIST input descriptions from even more abstract system descriptions have been developed, including the Table Oriented Translator to the ASSIST Language (TOTAL) (ref. 8) and the Reliability Estimation Testbed (REST) (ref. 9).

NASA Langley Research Center
Hampton, VA 23681-0001
April 7, 1995

Appendix A

Expression Precedence

This appendix defines the order of precedence used by ASSIST in evaluating mathematical expressions. The operator precedence is shown in the following table, which is sorted from highest precedence to lowest precedence.

Table A1. Order of Precedence Used by ASSIST for Mathematical Expressions

Precedence	Category	Operator	Associativity	Operand type	Result
Highest	Indexing	[ ]	Left to right	Any[int]	Same
	Grouping	( )	Left to right	Any	Same
	Binary	^	Right to left	Real ^ int	Real
	Unary	-	Right to left	Real,int	Same
	Unary	NOT	Right to left	Boolean	Boolean
	Binary	**	Right to left	Real,int	Real,int
	Binary	*,/,DIV,MOD,CYC	Left to right	Real(*,/),int	Same(*), real(/), int
	Binary	+,–	Left to right	Real,int	Same
	Binary	<,>,<=,>=,=,<>	Left to right	Real,int	Boolean
	Binary	==	Left to right	Boolean	Boolean
	Binary	AND,&	Left to right	Boolean	Boolean
Lowest	Binary	OR,|,XOR	Left to right	Boolean	Boolean
N/A	Rate dest.	++,--	N/A	State-space-var.	Same

A list of sample expressions follows. Each expression is followed by an arrow and a description of the ASSIST interpretation of the expression.

The dash is used for both unary negation and binary subtraction:

```
5 * – 6 ** 8  →  (5) * ((–6)**8)
  5 – 6 ** 8  →  (5) – (6**8)
```

The following are legal:

```
          NP MOD NFP + I  →  (NP MOD NFP) + I
          I + NP MOD NFP  →  I + (NP MOD NFP)
          I + NP DIV NFP  →  I + (NP DIV NFP)

        PSI * (2.0 + MU)  →  PSI * (2.0 + MU)
     PSI ** MU ** OMEGA   →  PSI ** (MU ** OMEGA)
       PSI / MU / OMEGA   →  (PSI / MU) / OMEGA
        PSI – MU –OMEGA   →  (PSI – MU) – OMEGA
           PSI ** – EPS   →  PSI ** (–EPS)
         PSI * MU ** 3    →  PSI * (MU ** 3)
   PSI + MU * XI + TAU    →  PSI + (MU * XI) + TAU
   PSI + MU < XI + TAU    →  (PSI + MU) < (XI + TAU)

  PSI < MU == XI >= TAU   →  (PSI < MU) == (XI >= TAU)†
TAU > MU & MU > RHO == TAU < RHO  →  (TAU > MU) & ((MU > RHO) == (TAU < RHO))†

    P AND Q OR P AND R    →  (P AND Q) OR (P AND R)
     P OR Q AND P OR R    →  P OR (Q AND P) OR R
```

†The == in this context is applied to Boolean values. The result will be TRUE whenever both sides are TRUE or both sides are FALSE.

In some instances, in order to avoid overflow/underflow, it is assumed that the user really intended to do real arithmetic. These instances are

```
      <integer> ** <negative-integer>   →  <real>
<integer> ** <large-positive-integer>   →  <real>
             <integer> / <integer>      →  <real>
```

For example

```
 2  **  -3  →  0.125
 10  **  9  →  1,000,000,000
10  **  11  →  100,000,000,000.0
       3/2  →  1.5
   9 DIV 2  →  4          (integer result)
   9 MOD 2  →  1          (integer result)
```

Appendix B

BNF Language Description

Appendix B gives a complete description of the ASSIST language syntax using the Backus-Naur Form (BNF) grammar.

```
                <program>  ::=  <setup-section>  <start-section>  <rule-section>

          <setup-section>  ::=  <setup-stat-seq>  <SPACE-stat>

          <start-section>  ::=  <start-stat-seq>  <START-stat>  <start-stat-seq>

           <rule-section>  ::=  <rule-stat-seq>


        <setup-stat-seq>  ::=  ε
                           |   <any-setup-sec-stat>  <setup-stat-seq>

        <start-stat-seq>  ::=  ε
                           |   <any-start-sec-stat>  <start-stat-seq>

         <rule-stat-seq>  ::=  <any-rule-sec-stat>
                           |   <any-rule-sec-stat>  <rule-stat-seq>


    <any-setup-sec-stat>  ::=  <global-stat>
                           |   <pre-rule-global-stat>

    <any-start-sec-stat>  ::=  <global-stat>
                           |   <pre-rule-global-stat>
                           |   <dep-variable-def>
                           |   <function-def>
                           |   <impl-function-def>

     <any-rule-sec-stat>  ::=  <global-stat>
                           |   <ASSERT-stat>
                           |   <DEATHIF-stat>
                           |   <PRUNEIF-stat>
                           |   <TRANTO-stat>
                           |   <IF-stat>
                           |   <FOR-stat>

<pre-rule-global-stat>  ::=  <quoted-SURE-stat>
                           |   <constant-def-stat>
                           |   <option-def-stat>
                           |   <INPUT-stat>

          <global-stat>  ::=  <debug-stat>
                           |   <command-option-stat>#
                           |   <empty-stat>

        <any-statement>  ::=  <any-setup-sec-stat>
                           |   <any-start-sec-stat>
                           |   <any-rule-sec-stat>
                           |   <SPACE-stat>
                           |   <START-stat>

        <reserved-word>  ::=  <sensitive-keyword>
                           |   <built-in-func-name>
```

```
                        |   <pre-defined-constant>
                        |   <descriptive-operator>
                        |   <statement-name>

   <sensitive-keyword>  ::= BY
                        |   FAST
                        |   THEN
                        |   ELSE
                        |   ENDIF
                        |   ENDFOR
                        |   WITH
                        |   OF
                        |   IN
                        |   ARRAY
                        |   ON
                        |   OFF
                        |   FULL
                        |   BOOLEAN

<pre-defined-constant>  ::= <option-def-name>
                        |   AUTOFAST
                        |   TRIMOMEGA
                        |   TRUE
                        |   FALSE

<descriptive-operator>  ::= AND
                        |   OR
                        |   NOT
                        |   MOD
                        |   CYC
                        |   DIV

      <statement-name>  ::= <option-def-name>
                        |   C_OPTION
                        |   DEBUG$
                        |   INPUT
                        |   SPACE
                        |   FUNCTION
                        |   IMPLICIT
                        |   VARIABLE
                        |   START
                        |   ASSERT
                        |   DEATHIF
                        |   PRUNEIF
                        |   PRUNIF
                        |   TRANTO
                        |   IF
                        |   FOR

     <option-def-name>  ::= ONEDEATH
                        |   COMMENT
                        |   ECHO
                        |   TRIM


   <constant-def-stat>  ::= <named-constant> = <const-var-def-clause> ;
<const-var-def-clause>  ::= <constant-def-clause>
                        |   BOOLEAN <constant-def-clause>
```

```
<constant-def-clause>  ::= <expr> ;
                         | ARRAY ( <expr-list-with-of> ) ;
                         | <single-sub-array> ;
                         | <double-sub-array> ;

<double-sub-array>     ::= [ <sub-array-list> ]

<sub-array-list>       ::= <single-sub-array> , <single-sub-array>
                         | <single-sub-array> , <sub-array-list>

<single-sub-array>     ::= [ <expr-list-with-of> ]

<option-def-stat>      ::= ONEDEATH <flag-status> ;
                         | COMMENT <flag-status> ;
                         | ECHO <flag-status> ;
                         | TRIM <flag-status> ;
                         | TRIM <flag-status> WITH <expr> ;

<INPUT-stat>           ::= INPUT <input-list> ;

<SPACE-stat>           ::= SPACE = <space-picture> ;


<function-def>         ::= FUNCTION <function-name> ( <function-parm-
                           list> ) = <expr> ;

<impl-function-def>    ::= IMPLICIT <impl-func-name> [ <impl-parm-
                           list> ] = <expr> ;
                         | IMPLICIT <impl-func-name>
                                  [ <impl-parm-list> ] ( <index-parm-
                           list> ) = <expr> ;

<dep-variable-def>     ::= VARIABLE <impl-func-name> [ <impl-parm-
                           list> ] = <const-var-def-clause> ;


<START-stat>           ::= START = <space-expression> ;>

<ASSERT-stat>          ::= ASSERT <boolean-expression> ;

<DEATHIF-stat>         ::= DEATHIF <boolean-expression> ;

<PRUNEIF-stat>         ::= PRUNEIF <boolean-expression> ;
                         | PRUNIF <boolean-expression> ;>

<TRANTO-stat>          ::= IF <boolean-expression> <TRANTO-clause> ;
                         | <TRANTO-clause> ;

<IF-stat>              ::= IF <boolean-expression> THEN
                             <rule-stat-seq>
                           ENDIF ;
                         | IF <boolean-expression> THEN
                             <rule-stat-seq>
                           ELSE
                             <rule-stat-seq>
                           ENDIF ;

<FOR-stat>             ::= FOR <for-range>
                               <rule-stat-seq>
                           ENDFOR ;
```

```
<quoted-SURE-stat>   ::=  "  <quot-text>"

<command-option-stat>  ::=  C_OPTION  <identifier>  ;
                        |   C_OPTION  <identifier>  =  <value>  ;

        <debug-stat>   ::=  DEBUG$  ;
                        |   DEBUG$  <identifier>  ;
        <empty-stat>   ::=  ;>

      <TRANTO-clause>  ::=  TRANTO  <space-destination-list>  BY
                            <rate-expression>
                        |   TRANTO  <space-expression>  BY
                            <rate-expression>

       <flag-status>   ::=  ε
                        |   OFF
                        |   ON
                        |   FULL
                        |   =0
                        |   =1
                        |   =2

        <input-list>   ::=  <input-item>
                        |   <input-item>  ,  <input-list>

        <input-item>   ::=  <named-constant>
                        |   <prompt-message>  :  <named-constant>
                        |   BOOLEAN  <named-constant>
                        |   BOOLEAN  <prompt-message>  :  <named-constant>

    <prompt-message>   ::=  "  <quot-text>  "

<function-parm-list>   ::=  ε
                        |   <identifier>
                        |   <identifier>  ,  <function-parm-list>

   <index-parm-list>   ::=  <identifier>
                        |   <identifier>  ,  <index-parm-list>

    <impl-parm-list>   ::=  <state-space-var>
                        |   <state-space-var>  ,  <impl-parm-list>

         <quot-text>   ::=  ε
                        |   <quot-text-char>  <quot-text>

    <quot-text-char>   ::=  <non-quote-ascii-char>

  <space-expression>   ::=  (  <space-expr-list>  )
```

```
<space-expr-list>  ::= <space-expr-item>
                     | <space-expr-item> , <space-expr-list>

<space-expr-item>  ::= <whole-or-boolean-expression>
                     | <whole-expression> OF <whole-or-boolean-
                       expression>
                     | <space-expression>


  <space-picture>  ::= ( <space-item-list> )

<space-item-list>  ::= <space-item>
                     | <space-item> , <space-item-list>

       <space-item>  ::= <state-space-var>
                     | <state-space-var> : <i-range>
                     | <state-space-var> : BOOLEAN
                     | <space-picture>
                     | <state-space-var> : ARRAY
                       [ <array-range> ]
                     | <state-space-var> : ARRAY
                       [ <array-range> OF <i-range>
                     | <state-space-var> : ARRAY
                       [ <array-range> OF BOOLEAN

      <array-range>  ::= <i-range>
                     | <i-range> , <i-range>


<space-destination-list>  ::= <space-destination>
                     | <space-destination> , <space-destination-list>

   <space-destination>  ::= <dest-adj-clause>

     <dest-adj-clause>  ::= <state-space-var> = <whole-or-boolean-
                       expression>
                     | <state-space-var> <inc-op>


        <for-range>  ::= <index-variable> = <whole-expression> ,
                       <whole-expression>†
                     | <index-variable> IN <set>

              <set>  ::= [ <set-range-list> ]

     <set-range-list>  ::= <i-range>
                     | <whole-expression>
                     | <i-range> , <i-range-list>

          <i-range>  ::= <lower-bound> .. <upper-bound>

      <lower-bound>  ::= <range-bound>

      <upper-bound>  ::= <range-bound>

      <range-bound>  ::= <whole-expression>§

  <rate-expression>  ::= <real-expression>
                     | < <real-expression> , <real-expression> >
```

```
                              |   < <real-expression> , <real-expression> ,
                                  <real-expression> >
                              |   FAST <real-expression>


          <expr-list-with-of> ::= <expr>
                              |   <whole-expression> OF <expr>
                              |   <expr> , <expr-list-with-of>
                              |   <whole-expression> OF <expr> ,
                                  <expr-list-with-of>

            <expression-list> ::= <expr>
                              |   <expr> , <expression-list>

          <built-in-expr-list> ::= <expr>
                              |   <expr> , <built-in-expr-list>
                              |   <wild-sub-array>
                              |   <wild-sub-array> , <built-in-expr-list>

              <wild-sub-array> ::= <named-constant> [ * , <whole-expression> ]
                              |   <named-constant> [ <whole-expression> , * ]>
                              |   <state-space-var> [ * , <whole-expression> ]
                              |   <state-space-var> [ <whole-expression> ,
                                  * ]>

                        <expr> ::= <real-expression>
                              |   <whole-expression>
                              |   <boolean-expression>

<whole-or-boolean-expression> ::= <whole-expression>
                              |   <boolean-expression>

            <whole-expression> ::= <integer-expression>

             <real-expression> ::= <numeric-expression>

          <integer-expression> ::= <numeric-expression>


          <boolean-expression> ::= <bool-term-expr>

              <bool-term-expr> ::= <bool-term>
                              |   <bool-term-expr> <or-op> <bool-term>

                   <bool-term> ::= <bool-factor>
                              |   <bool-term> <and-op> <bool-factor>

                 <bool-factor> ::= <bool-item>
                              |   <bool-item> == <bool-item>

                   <bool-item> ::= <numeric-comparison>
                              |   <simple-bool-item>

          <numeric-comparison> ::= <whole-expression> <relation>
                                   <whole-expression>

            <simple-bool-item> ::= <non-index-single-item>
                              |   <truth-value>
                              |   <boolean-function-invocation>
```

```
                    |   ( <boolean-expression> )
                    |   NOT <simple-bool-item>


          <or-op>   ::= OR
                    |   |
                    |   XOR

         <and-op>   ::= AND
                    |   &

        <relation>  ::= <inequality-relation>
                    |   <equality-relation>

<inequality-relation> ::= >
                    |   <
                    |   >=
                    |   <=

 <equality-relation> ::= <>
                    |   =


<numeric-expression> ::= <term-expr>

        <term-expr> ::= <term>
                    |   <term-expr> <add-op> <term>

            <term>  ::= <factor>
                    |   <term> <mpy-op> <factor>

          <factor>  ::= <numeric-item>
                    |   <numeric-item> <pow-op> <factor>

     <numeric-item> ::= <bin-numeric-item>
                    |   <sign-op> <numeric-item>

<bin-numeric-item>  ::= <non-index-single-item>
                    |   <index-variable>
                    |   <unsigned-value>
                    |   <named-constant> <cat-op> <bin-numeric-item>
                    |   <numeric-function-invocation>
                    |   ( <numeric-expression> )


          <add-op>  ::= +
                    |   -

          <mpy-op>  ::= *
                    |   /
                    |   MOD
                    |   CYC
                    |   DIV

          <pow-op>  ::= **

         <sign-op>  ::= -
```

```
                    <inc-op>  ::=  ++
                                |  --

                    <cat-op>  ::=  ^


<boolean-function-invocation>  ::=  <function-invocation>

<numeric-function-invocation>  ::=  <function-invocation>

        <function-invocation>  ::=  <impl-func-name>
                                |   <function-name>  (  <expression-list>  )
                                |   <built-in-name>  (  <built-in-expr-list>  )


      <non-index-single-item>  ::=  <named-constant>
                                |   <named-constant>  [  <whole-expression>  ]
                                |   <named-constant>  [  <whole-expression>
                                    <whole-expression>  ]
                                |   <state-space-var>
                                |   <state-space-var>  [  <whole-expression>  ]
                                |   <state-space-var>  [  <whole-expression>
                                    <whole-expression>  ]


             <function-name>  ::=  <identifier>

            <impl-func-name>  ::=  <identifier>

             <built-in-name>  ::=  SQRT     |  EXP      |  LN
                                |  SIN      |  COS      |  TAN
                                |  ARCSIN   |  ARCCOS   |  ARCTAN
                                |  FACT     |  SUM      |  COUNT
                                |  COMB     |  PERM     |  ABS
                                |  ANY      |  ALL      |  SIZE
                                |  MIN      |  MAX


               <truth-value>  ::=  FALSE
                                |  TRUE

                   <comment>  ::=  (*  <text>  *)
                                |  {  <text>  }

                     <under>  ::=  -

                    <dollar>  ::=  $

                    <E-char>  ::=  E  |  e  |  D  |  d

                    <letter>  ::=  A  |  B  |  C  |  D  |  E
                                |  F  |  G  |  H  |  I  |  J
                                |  K  |  L  |  M  |  N  |  O
                                |  P  |  Q  |  R  |  S  |  T
                                |  U  |  V  |  W  |  X  |  Y
                                |  Z
                                |  a  |  b  |  c  |  d  |  e
                                |  f  |  g  |  h  |  i  |  j
                                |  k  |  l  |  m  |  n  |  o
```

```
                                |   p  |q  |r  |s  |t
                                |   u  |v  |w  |x  |y
                                |   z

                      <digit>   ::= 0  |1  |2  |3  |4
                                |   5  |6  |7  |8  |9

                 <ident-char>   ::= <letter>
                                |   <digit>
                                |   <under>
                                |   <dollar>‡

                 <identifier>   ::= <letter>
                                |   <letter>  <ident-rest>

                 <ident-rest>   ::= <ident-char>
                                |   <ident-char>  <ident-rest>

     <unsigned-integer-value>   ::= <digit>
                                |   <digit>  <unsigned-integer-value>

        <unsigned-real-value>   ::= <unsigned-integer-value>  .  <unsigned-integer-
value>
                                |   <unsigned-integer-value>  .  <unsigned-integer-
value>  <exponent-value>

             <exponent-value>   ::= <E-char>  <sign-op>  <unsigned-integer-value>
                                |   <E-char>  <unsigned-integer-value>

             <named-constant>   ::= <identifier>

            <state-space-var>   ::= <identifier>

             <index-variable>   ::= <identifier>
```

The C_OPTION statement can be repeated but will usually precede any other statements.
† This syntax is obsolete at revision 7.0—its use will result in a warning message.
‡ All identifiers with dollar signs are reserved by the ASSIST language.
§ Although lower and upper bounds can take on values between 0 and 32767, their difference must be no more than 255.

Appendix C

Errors/Warnings Detected

The ASSIST language checks for errors in the input file. Appropriate error or warning messages are displayed depending upon the severity of the problem detected. The following conditions are tested. Some are petty warnings and are not reported unless **"/WARN=ALL"** is specified on the VMS command line or **"-WARN=ALL"** is specified on the UNIX command line. Some potential problems that are detected are reported as both errors and warnings, depending upon the context in which the problems were encountered.

Note that after detecting an error in a statement, ASSIST makes a best-guess attempt to recover enough so that it can continue to find more errors without generating too many extraneous error messages. Sometimes it is not possible to do this. If a line contains several error messages and most of them do not seem to apply, fix the ones that do apply. The other errors, especially when they follow the ones that do apply, will probably disappear when the fixes are made.

C.1. Listing of Detected Errors

An alphabetical listing of all error messages follows:

- `AN INTEGER EXPRESSION MUST PRECEDE THE WORD "OF":` ***token*** ⇒
  This error indicates that something other than an integer expression preceded the word "OF" during repetition (usually in a `START` statement). It is usually caused by the presence of a real number but can also be caused by an ill-formed expression.

- `ARCCOS(X) ARGUMENT MUST BE -1.0 <= X <= 1.0:` ***real*** `(ABS)> 1.0, (expression on line#`***linenumber***`)` ⇒
  This error indicates that, during rule generation, an expression that was passed to the built-in function `ARCCOS` evaluated to ***real***, a number greater in absolute magnitude than 1.0. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- `ARCCOS(X) ARGUMENT MUST BE -1.0 <= X <= 1.0: (expression on line#`***linenumber***`)` ⇒
  This error indicates that, during rule generation, an expression that was passed to the built-in function `ARCCOS` evaluated to a number greater in absolute magnitude than 1.0. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- `ARCSIN(X) ARGUMENT MUST BE -1.0 <= X <= 1.0:` ***real*** `(ABS)> 1.0, (expression on line#`***linenumber***`)` ⇒
  This error indicates that, during rule generation, an expression that was passed to the built-in function `ARCSIN` evaluated to ***real***, a number greater in absolute magnitude than 1.0. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- `ARCSIN(X) ARGUMENT MUST BE -1.0 <= X <= 1.0: (expression on line#`***linenumber***`)` ⇒
  This error indicates that, during rule generation, an expression that was passed to the built-in function `ARCSIN` evaluated to a number greater in absolute magnitude than 1.0. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC INTEGER OVERFLOW: *int* + *int*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the sum (or difference) of two integers exceeded the maximum or minimum value allowed by the hardware. Most models use very small integers. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC INTEGER OVERFLOW: *int* * *int*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the product of two integers exceeded the maximum or minimum value allowed by the hardware. Most models use very small integers. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC REAL OVERFLOW: (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, a value in a real expression exceeded the maximum or minimum value allowed by the hardware. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC REAL OVERFLOW: *real* + *real*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the sum (or difference) of two real numbers exceeded the maximum or minimum value allowed by the hardware. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC REAL OVERFLOW: *real* * *real*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the product of two real numbers exceeded the maximum or minimum value allowed by the hardware. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARITHMETIC OPERATOR IN A BOOLEAN EXPRESSION: *op* ⇒
  This error indicates that the user tried to do arithmetic on Boolean quantities. If the user desires to do arithmetic with the ordinal value of a Boolean, then the COUNT function must be used to convert the Boolean to an integer. Arithmetic on Booleans is otherwise disallowed.

- ARRAY DIMENSION OUT OF BOUNDS: (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the expression for a subscript of an array evaluated to a value that was not within the declared range. For example, if the array is bounded 5..11 and referenced with an expression evaluating to 12, then this error will occur. The user should examine the log file and check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ARRAY DIMENSION OUT OF BOUNDS: *token* ⇒
  This error indicates that, during parsing, the constant expression for a subscript of an array evaluated to a value that was not within the declared range. For example, if the array is bounded 5..11 and referenced with an expression evaluating to 12, then this error will occur. The user should check the expression for mistakes or increase the range of the array.

- ARRAY DIMENSION(S) MISSING: *token* ⇒
  This error indicates that the key word ARRAY was used but no subscript bounds were given inside brackets. The user may have forgotten the brackets entirely or may have used brackets but omitted an expression between them. The error could also be caused by an extraneous comma.

- ARRAY SUBSCRIPTS MUST BE ENCLOSED IN [ ]: *id* ⇒
  This error indicates that an array subscript was given but no square brackets were found. This error usually occurs when an array is referenced with parentheses instead of square brackets. The user should change the ( ) to [ ].

- ATTEMPT TO DEFINE RECURSIVE FUNCTION/IMPLICIT DISALLOWED: *num* ⇒
  This error indicates that a FUNCTION or IMPLICIT made reference to itself. For example, "FUNCTION FOO(X) = FOO(12) + X;" is illegal because the function FOO references itself by computing FOO(12) in the body of the function definition. The user should rethink the problem and change the function body.

- ATTEMPT TO TAKE LN OF NEGATIVE NUMBER: (expression on line#*line-number*) ⇒
  This error indicates that, during rule generation, an illegal attempt to compute the natural logarithm of a negative (or zero) number was made. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ATTEMPT TO TAKE LN OF NEGATIVE NUMBER: *real* <= 0.000000000, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, an illegal attempt to compute the natural logarithm of a negative (or zero) number was made. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ATTEMPT TO TAKE SQRT OF NEGATIVE NUMBER: *real* < 0.0000000000, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, an illegal attempt to compute the square root of a negative (or zero) number was made. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- ATTEMPT TO TAKE TAN() AT SINGULAR POINT (PI/2,3*PI/2,ETC): (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, an illegal attempt was made to take the tangent of a number near a singular point such as $\frac{\pi}{2}, \frac{3 \cdot \pi}{2}$. The user should examine the log file to check the expression for mistakes and check the ASSIST input file rule section for correctness.

- BAD TYPE IN A NUMERIC EXPRESSION: *id* ⇒
  This error indicates that a non-numeric quantity occurred in a numeric expression. It usually appears when Booleans are found when numeric expressions are expected.

- BOOLEAN ITEM EXPECTED: *token* ⇒
  This error indicates that something other than a Boolean item was encountered when the syntax for ASSIST required a Boolean item. The token fund instead of the Boolean item is echoed.

- BOOLEAN ITEM EXPECTED: *token* (BUILT-IN FUNCTION *func*) ⇒
  This error indicates that something other than a Boolean item was encountered in the parameter list for a built-in function such as COUNT when the syntax for ASSIST required a Boolean item. The token found instead of the Boolean item is echoed.

- BOOLEAN OPERATOR IN AN ARITHMETIC EXPRESSION: *op* ⇒
  This error indicates that something other than a Boolean operator was encountered in an arithmetic expression when the syntax for ASSIST required a Boolean item. The token found instead of the Boolean item is echoed. A relational operator was probably found in a numeric expression. The operator that is illegal in the expression is echoed.

- BOOLEAN VALUE EXPECTED: *token* ⇒
  This error indicates that the resulting type of an expression was not a Boolean. The token at the end of the expression in error is echoed.

- BUILT-IN FUNCTION SIZE REQUIRES AN ARRAY: *token* ⇒
  This error indicates that something other than an array was encountered as a parameter to the SIZE function when the syntax for ASSIST required an array. The token found instead of the array is echoed.

- CALLING PARAMETERS NOT ALLOWED ON IMPLICIT REFERENCES: *id* ⇒

  This error indicates that an IMPLICIT function, which was declared without a parameter list, was referenced with a passed parameter list.

- CANNOT MIX REAL NUMBERS INTO INTEGER ARRAY CONSTANT. ⇒

  This error indicates that the user was in the process of defining a constant integer array when a real number was encountered. The ASSIST file determines the type of array from the presence or absence of a period in the first element of the array. If the array is a real array with some integer values, then the first element in the array must have a decimal point. For example, "FOO = [7.0, 11, 8.5];". The decimal can be omitted for any integer elements in the real array unless the integer element comes first. If the array is an integer array, then no decimal points are allowed in any of the elements.

- CANNOT OPEN FILE. PLEASE CONTACT SYSTEM MANAGER: *foo.ast* ⇒

  This error indicates that there is probably something wrong with the disk drive since ASSIST could not open the input file. The system manager might have the disk drive dismounted or otherwise unavailable. ASSIST will usually tell the user why it cannot open a file. If it does not, the system manager would be aware of any unusual circumstances that would affect disk drive access.

- CANNOT RAISE A NEGATIVE NUMBER TO A REAL POWER: *real* ** *real*, (expression on line#*linenumber*) ⇒

  This error indicates that, during rule generation, an attempt was made to raise a negative number to a real power. If the power is something like "** 2.0" then it should be changed to "** 2".

- COLON EXPECTED: *token* ⇒

  This error indicates that something other than a colon was encountered when the syntax for ASSIST requires a colon. The token found instead of the colon is echoed.

- COMMA EXPECTED: *token* ⇒

  This error indicates that something other than a comma was encountered when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed.

- COMMA EXPECTED: *token* (FUNCTION *func*) ⇒

  This error indicates that something other than a comma was encountered in the parameter list for a FUNCTION when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed.

- COMMA EXPECTED: *token* (IMPLICIT *impl*) ⇒

  This error indicates that something other than a comma was encountered in the parameter list for an IMPLICIT when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed.

- COMMA EXPECTED: *func* REQUIRES 1-*num* PARAMETERS AND NO FEWER. ⇒

  This error indicates that something other than a comma was encountered in the parameter list for a built-in function when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed. This error usually occurs when a list function, such as SUM, COUNT, MIN, MAX, ANY or ALL, is invoked without any parameters.

- COMMA EXPECTED: *func* REQUIRES EXACTLY *num* PARAMETERS AND NO FEWER. ⇒

  This error indicates that something other than a comma was encountered in the parameter list for a built-in function when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed. This error usually occurs when one or more parameters for the named function are missing.

- COMMAND LINE OPTION TOO BIG: "-wid=nnn", (limit=256) ⇒
  This error indicates that the user tried to specify the maximum width of an input file line to be greater than 256. Input lines are usually limited to wid=80, or 79 characters. The user can increase the allowed width, but not to a value greater than 256.

- COMMAND LINE OPTION TOO BIG: "/wid=nnn", (limit=256) ⇒
  This error is the VMS version of the previous error message.

- COMMAND LINE OPTION TOO TINY: "-wid=nnn", (limit=38) ⇒
  This error indicates that the user tried to specify the maximum width of an input file line to be less than 38. Input lines are usually limited to wid=80, or 79 characters. The user can decrease the allowed width, but not to a value less than 38.

- COMMAND LINE OPTION TOO TINY: "/wid=nnn", (limit=38) ⇒
  This error is the VMS version of the previous error message.

- COMMAND LINE OPTION VALUE TOO BIG, MAX-ALLOWED = 32767. ⇒
  This error indicates that the value entered for a command-line option was bigger than could be parsed. The user should use a smaller value.

- CONSTANT IDENTIFIER NAME NOT UNIQUE TO FIRST 12 CHARS. SURE WILL NOT BE ABLE TO SOLVE THIS MODEL: *id* ⇒
  This error indicates that two different named constants have different names in ASSIST but will have the same name in SURE due to truncation to 12 characters by SURE. This message occurs only for named numeric constants since Boolean constants are not written to the model output file. The error is usually caused when a constant array is declared. For each item in a constant array, a constant scalar name is generated by appending the subscript(s) used to index the array to the end of the array name. For example, in the case of the doubly subscripted array named "LAMBDA_REC", identifiers such as "LAMBDA_REC_4_11" will be generated. These identifiers are not unique to the first 12 characters. To solve this problem, use a shorter array name.

- DISK QUOTA EXCEEDED. PLEASE CONTACT SYSTEM MANAGER: *foo.ext* ⇒
  This error indicates that there is not enough disk space left to open or write to the specified file. When a user gets this message, it is usually because the user has a lot of scratch files in the directory. Files that are no longer required should be deleted. VMS users can use the PURGE command to delete multiple versions of the same file. If, after cleaning up, the error still occurs, then the system manager should be contacted to get the user's quota increased.

- EMPTY LIST SPECIFIED: *id* ⇒
  This error indicates that the user followed a left parenthesis by a right parenthesis or a left bracket by a right bracket. It could also indicate that an INPUT statement did not specify any constants to be input, or that a SPACE statement did not have any state-space variables, or that a TRANTO did not list any destination expressions. The enclosed text may have been commented out as in "[(* 1,2 *)]".

- EOF REACHED BEFORE COMMENT TERMINATED: ("(*" on line#nnn) ⇒
  This error indicates that a comment starting on the specified line was never terminated with a matching "*)".

- EOF REACHED BEFORE COMMENT TERMINATED: ("{" on line#nnn) ⇒
  This error indicates that a comment starting on the specified line was never terminated with a matching "}".

- EOF REACHED BEFORE NESTED RULE SEQUENCE TERMINATED: ELSE OR ENDIF MISSING. ⇒
  This error indicates that no matching ELSE or ENDIF was found for a THEN. The user should use the **–xref** option to generate a cross-reference map on the log file. This cross-reference map can be

used to help determine if the user's indentation style is inconsistent and can help the user to determine where the missing ELSE or ENDIF belongs.

- EOF REACHED BEFORE NESTED RULE SEQUENCE TERMINATED: ENDFOR MISSING. ⇒
  This error indicates that no matching ENDFOR was found for a FOR. The user should use the **-xref** option to generate a cross-reference map on the log file. This cross-reference map can be used to help determine if the user's indentation style is inconsistent and can help the user to determine where the missing ENDFOR belongs.

- EOF REACHED BEFORE QUOTED TEXT TERMINATED: (" on line#nnn) ⇒
  This error indicates that the matching quote to end quoted text on the specified line is missing. This usually indicates that a prompt message in an INPUT statement or a SURE statement was never completed due to omission of the terminating quote character.

- EQUAL SIGN "=" EXPECTED: *token* ⇒
  This error indicates that something other than an equals sign was encountered when the syntax for ASSIST requires an equals sign. The token found instead of the equal sign is echoed.

- EXP(X) VALUE IS TOO BIG: *real* > *real*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, the value of "*x*" for exp^x was too big. The value passed for "*x*" is printed to the left of the > and the maximum value allowed is printed to the right of the >.

- EXPECT ON/OFF/FULL or =#: *id* ⇒
  This error indicates that an option definition name was followed by something other than one of the legal choices listed in the message. The user may have tried to redefine an option to be a named constant or a state-space variable. Option names can only appear in option definition statements.

- EXPRESSION OPERAND LIST OVERFLOW. SPECIFY -O OPTION: *id* ⇒
  This error indicates that a very long expression was being parsed. The expression was longer than ever anticipated. If the expression cannot be simplified, then increase the allowable size with the -o=*num* command-line option or in a "C_OPTION O=*num* ;" statement as the first line of the input file. The default is -o=100 for workstations and -o=50 for the IBM PC.

- EXPRESSION OPERAND LIST OVERFLOW. SPECIFY /O OPTION: *id* ⇒
  This error is the VMS equivalent of the previous error.

- EXPRESSION REQUIRED BUT WAS OMITTED: *token* ⇒
  This error indicates that something other than an expression was encountered when the syntax for ASSIST requires an expression. The token found instead of the expression is echoed.

- FILE NAME TOO LONG. ⇒
  This error indicates that the name of at least one of the ASSIST file names is too long. Remember that the longest file name that can be used for an ASSIST input file name ("*.ast*" file) is one less than the longest file name allowed by the operating system on systems that allow four character extents (such as UNIX and VMS).

- FILE NOT FOUND: *foo.ast* ⇒
  This error indicates that the specified input file does not exist. The user probably misspelled the file name.

- FUNCTION BODY STORAGE OVERFLOW. SPECIFY -B OPTION: *id* ⇒
  This error indicates that there were very many or very large FUNCTION and IMPLICIT functions. There was not enough body token storage allocated. To fix the problem, try to use more FUNCTIONs and IMPLICITs and eliminate common subexpressions. If that does not help, then

increase the body storage on the command line or in a "C_OPTION B=*num* ;" statement as the first line of the input file. The default is –b=1024 for workstations and –b=256 for the IBM PC.

- FUNCTION BODY STORAGE OVERFLOW. SPECIFY /B OPTION: *id* ⇒
  This error is the VMS equivalent of the previous error.

- FUNCTION/IMPLICIT NESTING LEVEL OVERFLOW. SPECIFY –NEST OPTION: *id* ⇒
  This error indicates that there were FUNCTIONs or IMPLICITs that referenced other FUNCTIONs or IMPLICITs and that the depth of the nested invocation exceeded the maximum allowed. The default is –n=16 for workstations and –n=8 for the IBM PC.

- FUNCTION/IMPLICIT NESTING LEVEL OVERFLOW. SPECIFY /NEST OPTION: *id* ⇒
  This error is the VMS equivalent of the previous error.

- IDENTIFIER ALREADY DEFINED. ⇒
  This error indicates that the user tried to redefine an identifier that had been defined previously. The identifier may have been used in the parameter list for a FUNCTION or IMPLICIT definition statement and is now being defined as a constant. Once an identifier is used in a parameter list, it may be reused in another parameter list or as a FOR index, but it may not be reused as a named constant. See sections 3.2.4, 3.2.3, and 12.2.7.3 for descriptions of the respective statements.

- IDENTIFIER EXPECTED: *token* ⇒
  This error indicates that something other than an identifier was encountered when the syntax for ASSIST requires an identifier. The token found instead of the identifier is echoed.

- IDENTIFIER NOT DEFINED: *id* ⇒
  This error indicates that an identifier was used before it was defined.

- IDENTIFIER OR LITERAL EXPECTED: *token* ⇒
  This error indicates that something other than an identifier or literal was encountered when the syntax for ASSIST requires an identifier or literal. The token found instead of the identifier or literal is echoed. The user probably has two arithmetic operations in a row without an item between them. For example, "*X* + * 3" is illegal because there is nothing between the "+" and the "*".

- IDENTIFIER TABLE OVERFLOW. SPECIFY -I,-N OPTIONS: (currently: -N=200 -I=400) ⇒
  This error indicates that the user is using more numbers or identifiers than the current symbol table can hold. The user must increase the table size with either the –n=*num* or –i=anynum command-line option or in a "C_OPTION I=*num* ;" or "C_OPTION N=*num* ;" statement as the first line of the input file. The default is –i=400 for workstations and –i=200 for the IBM PC. The default is –n=200 for workstations and –n=50 for the IBM PC.

- IDENTIFIER TABLE OVERFLOW. SPECIFY /I,/N OPTIONS: (currently: /N=200 /I=400) ⇒
  This error is the VMS equivalent of the previous error.

- INTEGER VALUE EXPECTED. ⇒
  This error indicates that the resulting type of an expression was not an integer. The token at the end of the expression in error is echoed.

- ILLEGAL CHARACTER: *char* ⇒
  This error indicates that a special character not allowed by the ASSIST syntax was encountered. Certain characters, including the exclamation mark ("!"), the percent sign ("%"), and all non-printing characters, are illegal in ASSIST.

- INFIX EXPRESSION LIST OVERFLOW. SPECIFY -O OPTION: *id* ⇒
  This error indicates that a very long expression was being parsed. The expression was longer than ever anticipated. If the expression cannot be simplified, then increase the allowable size with

the –o=num command-line option or in a "C_OPTION 0=*num* ;" statement as the first line of the input file. The default is **–o=100** for workstations and **–o=50** for the IBM PC.

- INFIX EXPRESSION LIST OVERFLOW. SPECIFY /O OPTION: *id* ⇒
  This error is the VMS equivalent of the previous error.

- INSUFFICIENT MEMORY FOR PARSE PHASE. ⇒
  This error indicates that the ASSIST input file is too big to parse. Since ASSIST will typically parse programs bigger than it can solve, it will probably be necessary to rethink the problem and make it simpler.

- INSUFFICIENT MEMORY FOR PARSE PHASE: *bytes* BYTES. OPERATING SYSTEM MALLOC LIMIT = *num* ⇒
  This error indicates that the total table size was bigger than ASSIST will allocate. The error always arises from large values for the **–b**, **–e**, **–o**, **–i**, **–n**, **–p**, **–nest**, **–pic**, and **–rule** options. Try decreasing the value specified on the command line for one or more of these options. The default is **–pic=100** for workstations and **–pic=100** for the IBM PC. The default is **–i=400** for workstations and **–i=200** for the IBM PC. The default is **–n=200** for workstations and **–n=50** for the IBM PC. The default is **–o=100** for workstations and **–o=50** for the IBM PC. The default is **–nest=16** for workstations and **–nest=8** for the IBM PC. The default is **–n=200** for workstations and **–n=50** for the IBM PC. The default is **–b=1024** for workstations and **–b=256** for the IBM PC. The default is **–e=100** for workstations and **–e=50** for the IBM PC. The default is **–p=64** for workstations and **–p=32** for the IBM PC. The default is **–rule=4096** for workstations and **–rule=1024** for the IBM PC.

- INSUFFICIENT MEMORY FOR PARSE PHASE: (Identifier table) ⇒
  This error indicates that the total table size was bigger than ASSIST will allocate. This error always arises from large values for the **–i** and **–n** options. Try decreasing the value specified on the command line for either or both of these options. The default is **–i=400** for workstations and **–i=200** for the IBM PC. The default is **–n=200** for workstations and **–n=50** for the IBM PC.

- INSUFFICIENT MEMORY FOR PARSE PHASE: (astparse storage) ⇒
  This error indicates that the total table size was bigger than ASSIST will allocate. This error always arises from large values for the **–e**, **–o**, **–p**, **–nest**, **–pic**, and **–rule** options. Try decreasing the value specified on the command line for one or more of these options. The default is **–pic=100** for workstations and **–pic=100** for the IBM PC. The default is **–o=100** for workstations and **–o=50** for the IBM PC. The default is **–nest=16** for workstations and **–nest=8** for the IBM PC. The default is **–n=200** for workstations and **–n=50** for the IBM PC. The default is **–e=100** for workstations and **–e=50** for the IBM PC. The default is **–p=64** for workstations and **–p=32** for the IBM PC. The default is **–rule=4096** for workstations and **–rule=1024** for the IBM PC.

- INSUFFICIENT MEMORY FOR PARSE PHASE: (parse library storage -- func body) ⇒
  This error is similar to the above error and arises from a large value for the **–b**. The default is **–b=1024** for workstations and **–b=256** for the IBM PC.

- INSUFFICIENT MEMORY FOR PARSE PHASE: (parse library storage) ⇒
  This error is similar to the "(astparse storage)" error above.

- INSUFFICIENT MEMORY FOR RULE GENERATION PHASE: (allstorage) ⇒
  This error is similar to the "(astparse storage)" error above.

- INSUFFICIENT MEMORY FOR RULE GENERATION PHASE: (data_ptr) ⇒
  This error is similar to the above error. Try pruning or trimming the model.

- INSUFFICIENT MEMORY FOR RULE GENERATION PHASE: (init_hash_function) ⇒
  This error is similar to the above error. The user probably asked for too large a number of hash buckets with the **-bc** option or too wide a bucket width with the **-bw** option. The defaults are **-bc=1009** and **-bw=5**.

- INSUFFICIENT MEMORY FOR RULE GENERATION PHASE: (link_to_a_brand_new_bucket) ⇒
  This error is similar to the above error. ASSIST was attempting to obtain more memory for a new state on the ready set but no more memory was available. Try pruning or trimming the model.

- INTEGER CYCLIC WRAP MODULO ZERO: ***int*** CYC ***int***, (expression on line#***linenumber***) ⇒
  This error indicates that, during rule generation, the user tried to divide by zero in order to obtain the cyclically wrapped integer remainder. This is illegal; do not do it.

- INTEGER DIVIDE BY ZERO: ***int*** DIV 0, (expression on line#***linenumber***) ⇒
  This error indicates that, during rule generation, the user tried to divide by zero in order to obtain the integer quotient. This is illegal; do not do it.

- INTEGER EXPRESSION EXPECTED: (expression on line#***linenumber***) ⇒
  This error indicates that, during rule generation, an expression that must evaluate to an integer evaluated to a real or Boolean. This error is fairly uncommon since ASSIST does a pretty thorough job of finding type mismatches during parsing.

- INTEGER EXPRESSION EXPECTED: MOD\CYC\DIV ⇒
  This error indicates that, during rule generation, an expression that must evaluate to an integer evaluated to a real or Boolean. This error is fairly uncommon since ASSIST does a pretty thorough job of finding type mismatches during parsing.

- INTEGER EXPRESSION EXPECTED: ***token*** ⇒
  This error indicates that, during parsing, an expression that must evaluate to an integer evaluated to a real or Boolean. It can also indicate that a strange token was found where an integer expression was expected.

- INTEGER EXPRESSION EXPECTED: ^ ⇒
  This error indicates that, during parsing, an expression that must evaluate to an integer evaluated to a real or Boolean. It probably means that the value following the concatenation operation was not an integer.

- INTEGER MODULO BY ZERO: ***int*** MOD ***int***, (expression on line#***linenumber***) ⇒
  This error indicates that, during rule generation, the user tried to divide by zero in order to obtain the integer remainder. This is illegal; do not do it.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: (unknown type for value_to_memory) ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Already has a type (lookup_ident)!!! ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Attempt to evaluate an expression in error. ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Attempt to evaluate the empty expression. ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: BAD BINARY OP ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: BAD TERNARY OP ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: BAD UNARY OP ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: BAD VARIABLE ARGUMENT LENGTH BUILT-IN OP ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: BUFFER OVERFLOW WHEN PARSING SPACE PICTURE. TRY USING 200 OR FEWER STATE-SPACE VARIABLES PER NESTING LEVEL. ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Expression evaluation stack overflow. Try simplifying expression via use of named constants or rebuild ASSIST with a larger value for EVAL_STACK_DIM. (expression on line#*num*) ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Not defined for SSV's/FUNCTION's/INTERNAL's (save_value_in_number_table) ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: Number of bits (*num*) in state space exceeds limit (8200) ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first. When the error message says, "Number of bits ... in state space exceeds limit", try simplifying the model by using fewer state-space variables.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: *id* ⇒
  This error indicates a problem so serious that ASSIST could not deal with it. Try fixing all other errors first.

- INVALID COMMAND LINE OPTION: -*opt* ⇒
  This error indicates that the user specified an unrecognizable command-line option. The name of the option was probably misspelled. Check the spelling against that listed for the desired option in appendix E.

- INVALID COMMAND LINE OPTION: /*opt* ⇒
  This error is the VMS equivalent of the previous error.

- INVALID NUMERIC VALUE: *id* ⇒
  This error indicates that a value was specified that is not in the range of valid values for the statement or construct in question. For example, in an option definition statement, only the values **"=0"**, **"=1"**, and **"=2"** are allowed, corresponding to "OFF", "ON", and "FULL", respectively.

- INVALID PROMOTION OF: <boolean> TO <integer> ⇒
  This error indicates that the user tried to mix Boolean and integer values in the same expression. The built-in function COUNT can be used to convert a Boolean to an integer and the relational operators can be used to convert a number to a Boolean. For example, "6 + COUNT(FLAG)" or "(I <> 0)", respectively.

- INVALID PROMOTION OF: <boolean> TO <real> ⇒
  This error indicates that the user tried to mix Boolean and real values in the same expression. The built-in function COUNT can be used to convert a Boolean to an integer and the relational operators can be used to convert a number to a Boolean. For example, "6 + COUNT(FLAG)" or "(I <> 0)", respectively.

- INVALID PROMOTION OF: <integer> TO <boolean> ⇒
  This error indicates that the user tried to mix Boolean and integer values in the same expression. The built-in function COUNT can be used to convert a Boolean to an integer and the relational operators can be used to convert a number to a Boolean. For example, "6 + COUNT(FLAG)" or "(I <> 0)", respectively.

- INVALID PROMOTION OF: <real> TO <boolean> ⇒
  This error indicates that the user tried to mix Boolean and real values in the same expression. The built-in function COUNT can be used to convert a Boolean to an integer and the relational operators can be used to convert a number to a Boolean. For example, "6 + COUNT(FLAG)" or "(I <> 0)", respectively.

- INVALID PROMOTION OF: <real> TO <integer> ⇒
  This error indicates that the user mixed real and integer numbers in an expression in such a way that the real number had to be converted to an integer. This resulted in the loss of the decimal portion of the real number. The user probably used real division ("/") instead of integer division ("DIV") somewhere in the expression.

- K CANNOT EXCEED N FOR COMB(N,K) AND PERM(N,K): *int* (COMB) k < n = *int*, (expression on line#*linenumber*) ⇒
  This error indicates that an attempt was made to compute the combinations of *n* things taken more at a time than there were things to take. The user probably specified the parameters to the built-in function in the wrong sequence. Check the expression very carefully to make sure.

- K CANNOT EXCEED N FOR COMB(N,K) AND PERM(N,K): *int* (PERM) k < n = *int*, (expression on line#*linenumber*) ⇒
  This error indicates that an attempt was made to compute the permutations of *n* things taken more at a time than there were things to take. The user probably specified the parameters to the built-in function in the wrong sequence. Check the expression very carefully to make sure.

- KEYWORD "BOOLEAN" MISSING FOR BOOLEAN CONSTANT INPUT OR DEFINITION. ⇒
  This error indicates that the expression pointed to evaluated to a Boolean value when a numeric expression was expected in an INPUT or constant definition statement. The user must tell ASSIST if a Boolean constant is being defined. For example, "FLAG = BOOLEAN X > Y;"

- KEYWORD "IN" IS MISSING: *id* ⇒
  This error indicates that the FOR statement requires the word "IN" before the set defining the FOR range. The user must say something like: "FOR III IN [1..10]".

- KEYWORD OR SEMICOLON EXPECTED: *id* ⇒
  This error indicates that something other than a key word or semicolon was encountered when the syntax for ASSIST requires a key word or semicolon. The token found instead of the key word or semicolon is echoed. There might be some extraneous text at the end of a DEBUG$, C_OPTION, or option definition statement.

- KEYWORD USED IN WRONG CONTEXT: DID YOU MEAN TO SAY "IMPLICIT"? ⇒
  This error indicates that the user used a reserved word in the wrong place. The user probably said "INTRINSIC" instead of "IMPLICIT".

- KEYWORD USED IN WRONG CONTEXT: WITH CLAUSE INVALID WHEN TRIM OFF. ⇒
  This error indicates that the user used the reserved word "WITH" in the wrong context. The user probably tried to specify a value for Ω_{trim} when trimming was off. See section 4.2 for more details on trimming.

- KEYWORD USED IN WRONG CONTEXT: *id* ⇒
  This error indicates that the user used a reserved word in the wrong place.

- LEFT "(" EXPECTED: *token* ⇒
  This error indicates that something other than a left parenthesis was encountered when the syntax for ASSIST requires a left parenthesis. The token found instead of the left parenthesis is echoed.

- LEFT "<" EXPECTED: NO MATCHING LEFT "<" FOR THIS ">" ⇒
  This error indicates that something other than a left-angle bracket was encountered in a rate expression when the syntax for ASSIST requires a left-angle bracket. The token found instead of the left-angle bracket is echoed.

- LEFT "[" EXPECTED: *token* ⇒
  This error indicates that something other than a left bracket was encountered in a rate expression when the syntax for ASSIST requires a left bracket. The token found instead of the left bracket is echoed.

- LOWER BOUND TO LEFT OF ".." RANGE MUST BE <= UPPER BOUND TO RIGHT: *min*..*max* ⇒
  This error indicates that a backwards range was specified in a SPACE statement, either as an array subscript range or a state-space variable value range. The user must specify the minimum value of the range before the maximum value for the range. For example, say "6..10" and not "10..6".

- MISSING TOKEN INSERTED BY PARSER:) ⇒
  This error indicates that something other than a closing parenthesis was encountered when the syntax for ASSIST requires a closing parenthesis. The token found instead of the closing parenthesis is echoed. The compiler was able to make an intelligent guess that the closing parenthesis was probably missing and that insertion of the missing closing parenthesis would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MISSING TOKEN INSERTED BY PARSER: , ⇒
  This error indicates that something other than a comma was encountered when the syntax for ASSIST requires a comma. The token found instead of the comma is echoed. The compiler was able to make an intelligent guess that the comma was probably missing and that insertion of the missing comma would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MISSING TOKEN INSERTED BY PARSER: ; ⇒
  This error indicates that something other than a semicolon was encountered when the syntax for ASSIST requires a semicolon. The token found instead of the semicolon is echoed. The compiler was able to make an intelligent guess that the semicolon was probably missing and that insertion

of the missing semicolon would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MISSING TOKEN INSERTED BY PARSER: = ⇒

  This error indicates that something other than an equals sign was encountered when the syntax for ASSIST requires an equals sign. The token found instead of the equals sign is echoed. The compiler was able to make an intelligent guess that the equals sign was probably missing and that insertion of the missing equals sign would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MISSING TOKEN INSERTED BY PARSER: OF ⇒

  This error indicates that something other than an "OF" key word was encountered when the syntax for ASSIST requires an "OF" key word. The token found instead of the "OF" key word is echoed. The compiler was able to make an intelligent guess that the "OF" key word was probably missing and that insertion of the missing "OF" key word would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MISSING TOKEN INSERTED BY PARSER:] ⇒

  This error indicates that something other than a closing bracket was encountered when the syntax for ASSIST requires a closing bracket. The token found instead of the closing bracket is echoed. The compiler was able to make an intelligent guess that the closing bracket was probably missing and that insertion of the missing closing bracket would probably prevent the detection of more extraneous errors. The insertion was therefore made, but the user will have to fix the problem before the model can be generated.

- MUST REBUILD USING THE HUGE MEMORY MODEL: ASSIST ⇒

  If ASSIST is compiled for use on a 386 or 387 IBM PC, then the huge memory model of the Microsoft C compiler must be used. Since SURE will not run on an IBM PC, ASSIST is not yet supported on the PC. If a user site ports ASSIST to the PC, be sure to run the production version of an ASSIST file through a tested SUN or VAX version before accepting the results. See your system manager.

- MUST SPECIFY THE WHOLE ARRAY, NOT SCALAR ELEMENT: ⇒

  Certain built-in functions operate on all the array elements. When this is the case, the user may not specify a subscript after the name of the array.

- NAME OF AN ARRAY EXPECTED: *id* ⇒

  Certain built-in functions operate on expressions or arrays. An attempt to pass in an implicit function with parameters as if it were an array will cause an error.

- NAMED CONSTANT EXPECTED: *token* BEFORE CONCATENATION OPERATOR ⇒

  This error indicates that something other than a named constant was encountered preceding a concatenation character when the syntax for ASSIST requires a named constant. The token found instead of the named constant is echoed. State-space variable names are not allowed before a concatenation operator. If a state-space array element must be referenced in a rate expression, use the array syntax with the index in square brackets.

- NEGATIVE VALUES NOT ALLOWED. USE A WHOLE NUMBER: *int* COMPUTED ⇒

  This error indicates that a negative number was found where a whole number (zero or positive) was expected. Negative integers are not allowed in the rule section except in rate expressions.

- NEGATIVE VALUES NOT ALLOWED. USE A WHOLE NUMBER: *int* IS THE COMPUTED VALUE. ⇒
  This error indicates that a negative number was computed for a constant expression where a whole number is required. This error usually indicates that one of the bounds on a state-space variable in the SPACE statement is negative or the repetition count to the left of the **OF** key word in a START statement is negative.

- NEGATIVE VALUES NOT ALLOWED. USE A WHOLE NUMBER: *int* (COMB) n < 0, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, a negative number was computed and passed to the built-in function **COMB**. The user needs to check the input file for an incorrect specification of the model. The expression that caused the problem can be most quickly located by examining the log file.

- NEGATIVE VALUES NOT ALLOWED. USE A WHOLE NUMBER: *int* (FACT) n < 0, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, a negative number was computed and passed to the built-in function **FACT**. The user needs to check the input file for an incorrect specification of the model. The expression that caused the problem can be most quickly located by examining the log file.

- NEGATIVE VALUES NOT ALLOWED. USE A WHOLE NUMBER: *int* (PERM) n < 0, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, a negative number was computed and passed to the built-in function **PERM**. The user needs to check the input file for an incorrect specification of the model. The expression that caused the problem can be most quickly located by examining the log file.

- NOT ALLOWED IN IMPLICIT DEFINITION BODY.
  ONLY STATE-SPACE VARIABLES, NAMED-CONSTANTS, OR LITERALS MAY BE INHERITED: *id* ⇒
  This error indicates that the indicated identifier could not be used in the body of an implicit definition. It was probably used as a parameter in a previous FUNCTION or IMPLICIT. It might be a state-space variable that was used in the body but not listed in the state-space variable list.

- NOT YET IMPLEMENTED. ⇒
  This error indicates that the user tried to use a future option of a pending version of ASSIST that is still under development and not yet supported. At release 7.0 of ASSIST, there are no features that generate this error.

- NUMBER OF ELEMENTS DOES NOT MATCH NUMBER OF ELEMENTS IN PREVIOUS ROW OF DOUBLY SUBSCRIPTED ARRAY CONSTANT: *id* ⇒
  This error indicates that a doubly subscripted array constant is being declared, but the rows of the table do not all have the same number of elements. If the rows are supposed to have different lengths, then pad the shorter rows with trailing zeros so that all rows have the same length.

- NUMBER OF ERRORS EXCEEDS LIMIT OF: *40* ⇒
  This error indicates that the maximum number of errors per input file has been exceeded. ASSIST will therefore no longer continue parsing the input file for additional errors/warnings. The limit can be increased with the **–el** command line option as in **"–el=50"**. The errors will ultimately have to be fixed anyway, so it is good practice to start correcting them even if the limit is being increased.

- NUMBER OF ERRORS/LINE EXCEEDS LIMIT OF: *5* ⇒
  This error indicates that the maximum number of errors per line has been exceeded. ASSIST will therefore no longer continue parsing the input file for additional errors/warnings. The limit can be

increased with the **-lel** command-line option as in **"-lel=10"**. The errors will ultimately have to be fixed anyway, so it is good practice to start correcting them even if the limit is being increased.

- NUMBER OF WARNINGS EXCEEDS LIMIT OF: *40* ⇒
  This error indicates that the maximum number of warnings per input file has been exceeded. ASSIST will therefore no longer continue parsing the input file for additional errors/warnings. The limit can be increased with the **-wl** command-line option as in **"-wl=50"**. The warnings will ultimately have to be fixed anyway, so it is good practice to start correcting them, even if the limit is being increased.

- NUMBER OF WARNINGS/LINE EXCEEDS LIMIT OF: *5* ⇒
  This error indicates that the maximum number of warnings per line has been exceeded. ASSIST will therefore no longer continue parsing the input file for additional errors/warnings. The limit can be increased with the **-lwl** command-line option as in **"-lwl=10"**. The warnings will ultimately have to be fixed anyway, so it is good practice to start correcting them, even if the limit is being increased.

- NUMBER TOO LONG OR VALUE TOO BIG: *int* IS THE COMPUTED VALUE. ⇒
  This error indicates that a number was computed for a constant expression and it was larger than allowed in its context. This error usually indicates that one of the bounds on a state-space variable in the SPACE statement is larger than 32767 or the repetition count to the left of the OF key word in a START statement is larger than 32767. State-space variables can have values as large as 32767, but the difference between the upper and lower bound on the range of values cannot exceed 255.

- NUMERIC ITEM EXPECTED: *id* (BUILT-IN FUNCTION *func*) ⇒
  This error indicates that something other than a numeric item was encountered in the parameter list for a built-in function such as SUM when the syntax for ASSIST requires a numeric item. The token found instead of the numeric item is echoed.

- ONLY ONE WILD SUBSCRIPT ALLOWED PER ARRAY REFERENCE: *id* ⇒
  This error indicates that a doubly wildcarded subarray was passed to a list function as in "SUM(FOO[*,*])". The syntax allows only for operation on a single row or a single column. If the whole table must be summed in both directions, then the correct syntax is "SUM(FOO)".

- ONLY READ PERMISSION CAN BE GRANTED: *foo.ast* ⇒
  This error indicates that the user does not have write permission to the directory in which the input file resides. The user should copy the input file to his own directory and run ASSIST there. If that does not work, contact the local system manager for help.

- ONLY STATE-SPACE VARIABLES MAY BE LISTED IN THE STATE-SPACE VARIABLE LIST: *id* ⇒
  This error indicates that the user attempted to list something other than a state-space variable name between the square brackets in an IMPLICIT definition statement. If the identifier is a named constant, it can be used without being listed. If the identifier is a function parameter, then it must be listed in the parameter list between the parentheses.

- PARAMETER COUNT OVERFLOW. SPECIFY -P OPTION: *id* ⇒
  This error indicates that the parameter (or state-space variable) list contained more than the maximum number of identifiers allowed. Since the default is fairly large, a function that exceeds it has so many parameters that it would probably be difficult to keep them straight. The best solution is to try to have a larger number of smaller functions with fewer parameters. The user can increase the parameter limit if necessary with the -p=*num* command-line option or in a "C_OPTION P=*num*;" statement as the first line of the input file. The default is **-p=64** for workstations and **-p=32** for the IBM PC.

- PARAMETER COUNT OVERFLOW. SPECIFY /P OPTION: *id* ⇒

  This error is the VMS equivalent of the previous error.

- PERMISSION DENIED. NO ACCESS GRANTED. PLEASE CONTACT SYSTEM MANAGER: *foo.ast* ⇒

  This error indicates that the user does not have permission to read the input file. The user should talk to the owner of the file about gaining permission or contact the local system manager for help.

- POSTFIX EXPRESSION STACK OVERFLOW. SPECIFY -O OPTION: COMMAND LINE OPTIONS TOO BIG. ⇒

  This error indicates that the user asked for more memory for the expression stack than allowed. Try asking for less. The default is **–o=100** for workstations and **–o=50** for the IBM PC.

- POSTFIX EXPRESSION STACK OVERFLOW. SPECIFY /O OPTION: COMMAND LINE OPTIONS TOO BIG. ⇒

  This error is the VMS equivalent of the previous error.

- POSTFIX EXPRESSION STACK OVERFLOW. SPECIFY -O OPTION: *id* ⇒

  This error indicates that a very long expression was being parsed. The expression was longer than ever anticipated. If the expression cannot be simplified, then increase the allowable size with the –o=*num* command-line option or in a "C_OPTION O=*num* ;" statement as the first line of the input file. The default is **–o=100** for workstations and **–o=50** for the IBM PC.

- POSTFIX EXPRESSION STACK OVERFLOW. SPECIFY /O OPTION: *id* ⇒

  This error is the VMS equivalent of the previous error.

- PRODUCT OF DIMENSION RANGES IS GREATER THAN 256: *id* ⇒

  This error indicates that the user declared a doubly subscripted array table with more than 256 total elements in it. Try using a smaller table size.

- PROGRAM MUST CONTAIN AT LEAST ONE TRANSITION: *id* ⇒

  This error indicates that the program did not contain any TRANTO statements. Every ASSIST input file must contain at least one transition.

- PROMPT MESSAGE IS TOO LONG. MESSAGE TRUNCATED: *message* ⇒

  This error indicates that the prompt message specified in an INPUT statement was longer than allowed. Prompt messages should be brief, yet clear. A prompt message may not have more than 90 characters.

- PROMPT STRING OR IDENTIFIER EXPECTED: *token* ⇒

  This error indicates that something other than a prompt string or identifier was encountered in an INPUT statement when the syntax for ASSIST requires a prompt string or identifier. The token found instead of the prompt string or identifier is echoed.

- QUITTING COMPILATION !!! ⇒

  This error indicates that ASSIST was unable to continue processing the input file because of an inability to correct for previous errors. The user should fix all known errors and this error will disappear.

- RANGE IS TOO WIDE. DIFFERENCE "UPPER-LOWER" LIMITED BY: 255 (*min*..*max*) ⇒

  This error indicates that the range specified by the user is too wide. Only 256 dimension slots are allowed, inclusive of both end points of the range. The user must make *min* and *max* closer together.

- RANGE IS TOO WIDE. DIFFERENCE "UPPER-LOWER" LIMITED BY: 32767 (*min*..*max*) ⇒
  This error indicates that the range specified by the user is too wide. Only 32767 values are allowed, inclusive of both end points of the range. The user must make *min* and *max* closer together.

- RATE EXPRESSION MUST BEGIN WITH KEYWORD "BY": *token* ⇒
  This error indicates that the destination clause of a TRANTO statement was finished but no "BY" followed the clause. This problem can be caused when one of the destination expressions is missing an operator, causing ASSIST to think that the destination clause is finished when in fact it really is not. The user should check the expression immediately prior to the designated token for validity.

- REAL DIVIDE BY ZERO: *real*/*real*, (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, an illegal attempt was made to divide by zero. The user should check the rule section for validity, especially in the vicinity of the flagged expression. Lines are numbered in the log file.

- REAL EXPRESSION EXPECTED: (expression on line#*linenumber*) ⇒
  This error indicates that, during rule generation, an expression was evaluated and it resulted in a nonreal value. This error is fairly uncommon since ASSIST does a pretty thorough job of finding type mismatches during parsing.

- REAL EXPRESSION EXPECTED: *token* ⇒
  This error indicates that, during parsing, the echoed token was found in a real expression when it does not belong there. Usually, the token is some kind of a relational operator, which would be valid only in Boolean expressions. If necessary, a Boolean expression can be changed to a numeric expression by counting it with the COUNT function.

- REAL NUMBERS NOT ALLOWED EXCEPT IN RATE EXPRESSIONS: *func* ⇒
  This error indicates that a real-valued built-in function, such as GAM, was found somewhere other than in a rate expression or constant definition.

- REAL NUMBERS NOT ALLOWED EXCEPT IN RATE EXPRESSIONS: *id* ⇒
  This error indicates that a real number was found somewhere other than in a rate expression or constant definition.

- REAL VALUE EXPECTED. ⇒
  This error indicates that the resulting type of an expression was not a real number. The token at the end of the erroneous expression is echoed.

- RELATIONAL OPERATOR MUST FOLLOW NUMERIC QUANTITY IN BOOLEAN EXPRESSION: *id* ⇒
  This error indicates that, when parsing a Boolean expression, a numeric expression came to an end but was not followed by a relational operator to compare it with some other numeric quantity. The user should fix the expression and rerun.

- RELATIONAL OPERATOR NOT ALLOWED IN A NUMERIC EXPRESSION: *id* ⇒
  This error indicates that, when parsing a numeric expression, a relational operator was encountered. The user should fix the expression and rerun.

- RIGHT ")" EXPECTED: *func* REQUIRES 1-*num* PARAMETERS AND NO MORE. ⇒
  This error indicates that something other than a right parenthesis was encountered in the parameter list for a built-in function when the syntax for ASSIST requires a right parenthesis. The token found instead of the right parenthesis is echoed. This error usually occurs when more parameters are passed to a built-in function than it is defined to handle.

- RIGHT ")" EXPECTED: *func* REQUIRES EXACTLY *num* PARAMETERS AND NO MORE. ⇒
  This error indicates that something other than a right parenthesis was encountered in the parameter list for a built-in function when the syntax for ASSIST requires a right parenthesis. The token found instead of the right parenthesis is echoed. This error usually occurs when more parameters are passed to a built-in function than it is defined to handle.

- RIGHT ")" EXPECTED: *id* ⇒
  This error indicates that something other than a right parenthesis was encountered when the syntax for ASSIST requires a right parenthesis. The token found instead of the right parenthesis is echoed.

- RIGHT ">" EXPECTED: *id* ⇒
  This error indicates that something other than a right-angle bracket was encountered when the syntax for ASSIST requires a right-angle bracket. The token found instead of the right-angle bracket is echoed. This usually indicates that the user is missing a right-angle bracket when White's Method is used to specify the transition rate expression in a TRANTO statement.

- RIGHT "]" EXPECTED: *id* ⇒
  This error indicates that something other than a right bracket was encountered when the syntax for ASSIST requires a right bracket. The token found instead of the right bracket is echoed.

- RULE SCRATCH STORAGE OVERFLOW. SPECIFY -RULE OPTION: *id* ⇒
  This error indicates that the user has more rule statements in series than allowed for at the nesting level where the error occurred. There may be too many rules in the rule section or just too many rules between an IF and an ELSE, or between a FOR and an ENDFOR. The storage can be increased with the **-rule** command-line option. The default is **-rule=4096** for workstations and **-rule=1024** for the IBM PC.

- SCALAR EXPECTED: *arrayname op* ⇒
  This error indicates that an array was found when a scalar was expected. The user was attempting to perform arithmetic on a whole array instead of just on one element of the array. The user probably forgot to specify a subscript in square brackets following the name of the array.

- SCALAR EXPECTED: *arrayname* ⇒
  This error indicates that an array was found when a scalar was expected. The user probably forgot to specify a subscript in square brackets following the name of the array.

- SCOPE OF IDENTIFIER IS INACTIVE: *id* ⇒
  This error indicates that an identifier was referenced but its use is no longer valid. It usually occurs when a FOR index is referenced after the matching ENDFOR. This error can also occur if an identifier name is referenced without redefinition after it has already been used in the parameter list for the definition of a prior FUNCTION or IMPLICIT.

- SCOPE OF IDENTIFIER IS INACTIVE: *id* (BUILT-IN FUNCTION *func*) ⇒
  This error indicates that an identifier was referenced but its use is no longer valid. It usually occurs when a FOR index is referenced after the matching ENDFOR, but the error can also occur if an identifier name is referenced without redefinition after it has already been used in the parameter list for the definition of a prior FUNCTION or IMPLICIT. The identifier in error was passed to the named built-in function.

- SCOPE OF IDENTIFIER IS STILL ACTIVE. ⇒
  This error indicates that the user tried to redefine an identifier when it was still being used for another purpose. This error usually means that two nested FOR constructs are using the same index variable name. The user should change the name of one index variable.

- SCRATCH EXPRESSION STORAGE OVERFLOW. SPECIFY -E OPTION: COMMAND LINE OPTIONS TOO BIG. ⇒

  This error indicates that the user asked for more memory for the expression storage than allowed. Try asking for less. The default is **-e=100** for workstations and **-e=50** for the IBM PC.

- SCRATCH EXPRESSION STORAGE OVERFLOW. SPECIFY -E OPTION: *id* ⇒

  This error indicates that the ASSIST program cannot hold all the expressions it needs at once. It usually indicates use of very many state-space variables at a single level in the SPACE statement, or it can indicate use of very many expressions in the rules nested between IF and ENDIF or between FOR and ENDFOR. The limit can be increased. The default is **-e=100** for workstations and **-e=50** for the IBM PC.

- SCRATCH EXPRESSION STORAGE OVERFLOW. SPECIFY /E OPTION: *id* ⇒

  This error is the VMS equivalent of the previous error.

- SEMICOLON ";" CHANGED TO COMMA "," BY PARSER. ⇒

  This error indicates that ASSIST found a semicolon but was able to guess that the user meant to use a comma instead. The change was made in an attempt to limit the number of extraneous errors during continued parsing of the statement in error.

- SEMICOLON EXPECTED: *id* ⇒

  This error indicates that something other than a semicolon was encountered when the syntax for ASSIST requires a semicolon. The token found instead of the semicolon is echoed. If the semicolon was missing before the end of the file, then the message "(end-of-file)" will be printed to the right of the colon.

- SKIPPING EXTRANEOUS TOKENS: *id* ⇒

  This error indicates that previous errors prevented continued parsing of part or all of the current statement. Fix the other errors in this statement and this error should disappear.

- SPACE STATEMENT IS MISSING. ⇒

  This error indicates that the first rule was encountered, yet the required SPACE statement was absent.

- SPACE STATEMENT OVERFLOW. SPECIFY -PIC OPTION: *id* ⇒

  This error indicates that there were more state-space variables at a single level in the SPACE statement than anticipated by ASSIST. The user should try to limit the number of state-space variables as much as is feasible. Unpruned models with a huge state space will seem like they run forever. The limit can be raised. The default is **-pic=100** for workstations and **-pic=100** for the IBM PC.

- SPACE STATEMENT OVERFLOW. SPECIFY /PIC OPTION: *id* ⇒

  This error is the VMS equivalent of the previous error.

- SPECIAL VMS ERROR NUMBER: *errornumber* ⇒

  This error indicates that an error specific to the VMS operating system was encountered.

- START STATEMENT IS MISSING. ⇒

  This error indicates that the first rule was encountered yet the required START statement was absent.

- STATE SPACE CANNOT BE EMPTY. AT LEAST ONE STATE-SPACE VARIABLE REQUIRED: *id* ⇒

  This error indicates that a SPACE statement was declared without any state-space variables. See section 3.2.1.6 for the correct syntax for the SPACE statement.

- STATE-SPACE VARIABLE DOES NOT HAVE A VALUE UNTIL AFTER PARSING IS COMPLETE AND RULE GENERATION HAS STARTED. ⇒
  This error indicates that a state-space variable was referenced in a context where a constant was required. It can occur in a constant definition statement, an INPUT statement, or in a START statement.

- STATE-SPACE VARIABLE EXPECTED: *token* ⇒
  This error indicates that something other than a state-space variable was encountered when the syntax for ASSIST requires a state-space variable. The token found instead of the state-space variable is echoed. It usually occurs in the destination clause of a TRANTO during an illegal attempt to change the value of an identifier other than a state-space variable.

- STATE-SPACE VARIABLE VALUE IS OUT OF RANGE: *num* (*min* to *max*) ⇒
  This error indicates that, during rule generation or during the parsing of a START statement, a state-space variable took on a value outside the range declared to be legal in the SPACE statement. Be sure to check for consistency between the SPACE, START, and all TRANTO statements. See sections 3.2.1.6, 3.2.6, and 3.2.7.1 for further details.

- STATEMENT EXPECTED: *token* ⇒
  This error indicates that ASSIST was looking for the beginning of a statement but found the token echoed to the right of the colon. All statements must begin with a reserved word except for the SURE statement, which begins with a quote, and the constant definition statement, which begins with an identifier followed by an equals sign.

- STATEMENT NOT VALID IN THIS SECTION: (constant definition of *id* = <expr>) ⇒
  Once the rule section begins, the user may no longer define any more constants because the rules are applied iteratively to all model states and are, by definition, nonconstant. If the definition is for a variable, then the user should consider the use of an IMPLICIT or FUNCTION definition. See section 3.2.7 for a description of the rule section and sections 3.2.3 and 3.2.4 for details on IMPLICIT and FUNCTION definitions, respectively.

- STATEMENT NOT VALID IN THIS SECTION: *id* ⇒
  This error indicates that a statement is out of sequence. Move the statement to its proper location and the error will disappear.

- STATEMENT OUT OF SEQUENCE ... NOT ALLOWED IN RULE SECTION. ⇒
  This error indicates that a statement is out of sequence. Move the statement to its proper location and the error will disappear. The setup, start, and rule sections are described in sections 3.2.1, 3.2.2, and 3.2.7, respectively. They are listed in the BNF description in appendix B.

- STATEMENT OUT OF SEQUENCE ... NOT ALLOWED IN SETUP SECTION. ⇒
  This error indicates that a statement is out of sequence. Move the statement to its proper location and the error will disappear. The setup, start, and rule sections are described in sections 3.2.1, 3.2.2, and 3.2.7, respectively. They are listed in the BNF description in appendix B.

- STATEMENT OUT OF SEQUENCE ... NOT ALLOWED IN START SECTION. ⇒
  This error indicates that a statement is out of sequence. Move the statement to its proper location and the error will disappear. The setup, start, and rule sections are described in sections 3.2.1, 3.2.2, and 3.2.7, respectively. They are listed in the BNF description in appendix B.

- SUBSCRIPT NOT ALLOWED ON SCALAR: *id* ⇒
  This error indicates that the identifier in question is being referenced as if it were an array even though it was not declared to be an array. Square brackets must be used to declare arrays. See section 3.1.5 for details on how to declare constant arrays. See section 3.2.1 for details on how to declare state-space variable arrays.

- THE REPETITION COUNT PRECEDING THIS "OF" IS TOO LARGE: *int* REPETITIONS IGNORED ⇒
  This error indicates that the repetition count before an "OF" clause is too large. It is probably either a typographical error or an erroneous expression.

- "THEN" OR "TRANTO" CLAUSE MISSING FOR THIS "IF": *id* ⇒
  This error indicates that an IF statement contained neither the THEN nor TRANTO key word. There are only two kinds of statements that begin with the key word "IF", namely the block IF and TRANTO statements. See sections 12.2.7.1 and 12.2.7.2 for details.

- THERE MUST BE EITHER TWO OR THREE EXPRESSIONS BETWEEN ANGLE BRACKETS IN A RATE EXPRESSION: *id* ⇒
  This error indicates that an incorrect number of expressions were listed when using White's Method to specify the transition rate expression in a TRANTO statement. See section 3.2.7.1 for more details.

- THERE MUST BE EXACTLY ONE EXPRESSION FOR A SLOW TRANSITION RATE: *id* ⇒
  This error indicates that an incorrect number of expressions were listed when specifying a slow transition rate in the rate expression for a TRANTO statement. If the user meant to give a mean and standard deviation, then angle brackets are required. If the user meant to have a slow transition rate, then an operator is probably missing from the expression or there may be an extraneous comma. See section 3.2.7.1 for more details.

- THIS FORM INVALID WITH "FAST" KEYWORD: *token* ⇒
  This error indicates that the user tried to mix the FAST key word with an incompatible form in the transition rate for a TRANTO statement. The user probably mixed the FAST key word with White's Method as in "BY FAST < *mean* , *sigma* >". Use the FAST key word or the angle-bracket syntax, but not both. See section 3.2.7 for more details.

- TOO FEW CALLING PARAMETERS. MORE EXPECTED: *token* (FUNCTION *func*) ⇒
  This error indicates that the FUNCTION reference had fewer parameters passed than were expected. Check for consistency between the reference on the line in error and the corresponding FUNCTION statement.

- TOO FEW CALLING PARAMETERS. MORE EXPECTED: *token* (IMPLICIT *impl*) ⇒
  This error indicates that the IMPLICIT reference had fewer parameters passed than were expected. Check for consistency between the reference on the line in error and the corresponding IMPLICIT statement.

- TOO FEW CALLING PARAMETERS. MORE EXPECTED: *token* ⇒
  This error indicates that a built-in function reference had fewer parameters passed than were expected. Check for consistency between the reference on the line in error and the definition of the built-in function being used. Section 3.1.8 details the different built-in functions.

- TOO FEW SUBSCRIPTS. THIS IS A DOUBLY SUBSCRIPTED ARRAY: *arrayname* ⇒
  This error indicates that a doubly subscripted array was referenced with fewer than two subscripts. Check for consistency with the declaration of the array.

- TOO MANY CALLING PARAMETERS. REMAINING IGNORED: *token* (FUNCTION *func*) ⇒
  This error indicates that the FUNCTION reference had more parameters passed than were expected. Check for consistency between the reference on the line in error and the corresponding FUNCTION statement.

- TOO MANY CALLING PARAMETERS. REMAINING IGNORED: *token* (IMPLICIT *impl*) ⇒
  This error indicates that the IMPLICIT reference had more parameters passed than were expected. Check for consistency between the reference on the line in error and the corresponding IMPLICIT statement.

- TOO MANY CALLING PARAMETERS. REMAINING IGNORED: *num* ⇒
  This error indicates that the built-in function reference had more parameters passed than were expected. See section 3.1.8 for details on the built-in functions.

- TOO MANY OPEN FILES. CANNOT OPEN. PLEASE CONTACT SYSTEM MANAGER: *filename* ⇒
  This error indicates that an attempt was made to open the file but too many files were already open. The user may have files open from suspended or background processes. Try logging out of all your processes and logging back in. If that does not work, or you get this error repeatedly, contact your system manager about having your limits increased.

- TOO MANY SUBSCRIPTS. ONLY SINGLY OR DOUBLY SUBSCRIPTED ARRAYS ARE ALLOWED: *id* ⇒
  This error indicates that an illegal attempt was made to declare an array with more than two dimensions. Triply subscripted arrays are not allowed by ASSIST.

- TOO MANY SUBSCRIPTS. THIS IS A SINGLY SUBSCRIPTED ARRAY: *arrayname* ⇒
  This error indicates that a singly subscripted array was referenced with more than one subscript. Check for consistency with the declaration of the array.

- TRIPLE BOOLEAN EQUALITY NOT SUPPORTED. USE "A==B AND B==C" INSTEAD OF "A==B==C", ETC: *token* ⇒
  This error indicates that the user tried to use triple Boolean equality without the use of the key word AND. The fix should be clear from the message.

- TRIPLE EQUALITY NOT SUPPORTED. USE "A=B AND B=C" INSTEAD OF "A=B=C", ETC: *token* ⇒
  This error indicates that the user tried to use triple equality without the use of the key word AND. The fix should be clear from the message.

- TRIPLE INEQUALITY NOT SUPPORTED. USE "A<B AND B<C" INSTEAD OF "A<B<C", ETC: *token* ⇒
  This error indicates that the user tried to use triple inequality without the use of the key word AND. The fix should be clear from the message.

- TYPE MISMATCH: *arrayname*[*num*] IS A BOOLEAN SSV. ⇒
  This error indicates that the user tried to do arithmetic with a Boolean state-space variable. This is usually undesirable, but in those instances when it is useful, the COUNT function is available to convert the Boolean state-space variable to an integer value.

- TYPE MISMATCH: *arrayname*[*num*] IS AN INTEGER SSV. ⇒
  This error indicates that the user tried to test an integer state-space variable as if it were a Boolean. This is usually undesirable, but in those instances when it is useful, the user can test for "*ssv* <> 0".

- TYPE MISMATCH: *id* ⇒
  This error indicates that two incompatible types were mixed in the same expression or the wrong value type was used for the context.

- TYPE MISMATCH: *id* IS A BOOLEAN SSV. ⇒
  This error indicates that the user tried to do arithmetic with a Boolean state-space variable. This is usually undesirable, but in those instances when it is useful, the COUNT function is available to convert the Boolean state-space variable to an integer value.

- TYPE MISMATCH: id IS AN INTEGER SSV. ⇒
  This error indicates that the user tried to test an integer state-space variable as if it were a Boolean. This is usually undesirable, but in those instances when it is useful, the user can test for "*ssv* <> 0".

- VALID ONLY IN RATE EXPRESSION: ^ (CONCATENATION) ⇒
  This error indicates that the concatenation operation is valid only in a rate expression. Concatenation can usually be replaced by specifying an array subscript inside square brackets. This should fix the problem.

- VARIABLE CANNOT BE REFERENCED IN BODY UNLESS LISTED IN STATE-SPACE VARIABLE LIST: *ssv* ⇒
  This error indicates that an attempt was made to use a state-space variable in the body of an IMPLICIT or FUNCTION definition without having declared it in the state-space variable list for an IMPLICIT. If the user is defining a FUNCTION, it will have to be converted to an IMPLICIT in order to reference the state-space variable. See sections 3.2.3 and 3.2.4 for details.

- VARIABLE STATEMENT CANNOT CONTAIN A FUNCTION PARAMETER LIST. CONSIDER USING AN IMPILICT FUNCTION: *token* ⇒
  This error indicates that a VARIABLE statement definition cannot have a parameter list following the state-space variable list.

- VARIABLES NOT ALLOWED IN CONSTANT DEFINITION EXPRESSION. CONSIDER THE USE OF A FUNCTION OR IMPLICIT: *id* ⇒
  This error indicates that a variable was referenced in a constant expression. It can occur in a constant definition statement, an INPUT statement, or in a START statement. Many input files built for older versions of ASSIST will get this error message because the old variable definition statement was replaced with the new IMPLICIT definition statement. See section 3.2.3 for details on the IMPLICIT statement.

- VARIABLES NOT ALLOWED IN FUNCTION DEFINITION BODY. ONLY NAMED CONSTANTS MAY BE INHERITED. ⇒
  This error indicates that the user tried to reference a variable from a function body. If the variable is a state-space variable, try using an IMPLICIT function definition instead. If the variable is a parameter from a previous FUNCTION or IMPLICIT, it was probably omitted from the parameter list for the new function.

- WILD SUBSCRIPT NOT ALLOWED EXCEPT IN CONTEXT OF APPLICABLE BUILT-IN FUNCTION: *id* ⇒
  This error indicates that an array was referenced with an asterisk instead of an array subscript in the wrong context. If the user meant to compute the subscript as the product of two quantities, an identifier or literal value must precede the asterisk. If the user meant for a wild subscript to be present, a function such as SUM or COUNT is probably missing.

C.2. Listing of Detected Warnings

The **–w=<level>** command-line option is used to specify the number of levels of warning reporting. All warnings at a level less than or equal to the requested number of reporting levels are printed to the standard error file and to the log file. The default is four levels of warning reporting.

A list of all warning levels in ASSIST appears in table C1.

Table C1. ASSIST Warning Levels

Level	Description
0	No warnings whatsoever. (Same as **–w=none**.)
1	The "Serious" level. These warnings are major. (Same as **–w=fewer**.)
2	The "Default" level. These warnings are usually worth noting.
3	The "Minor" level. These warnings are less major. (Same as **–w=all**.)

An alphabetical listing of all warning messages follows:

- ASSERTION FAILED: *token* ⇒

  This warning indicates that an invariant or other assertion failed because of a certain model state. The state node for the current state and the log file line number corresponding to the failed ASSERT condition are printed following the warning message. This serious warning message appears when one or more levels of error reporting are requested.

- COMMENT OPTION TURNED BACK OFF DUE TO EXCESSIVE STATE-SPACE VARIABLE COUNT: *id* ⇒

  This warning indicates that, since the state space is so large, ASSIST refuses to allow the COMMENT option to be ON. To get rid of this message, add the statement "COMMENT OFF;" to the ASSIST input file anywhere before the SPACE statement. This warning message appears when two or more levels of error reporting are requested.

- ELLIPSIS ".." AND UPPER BOUND ARE MISSING: USING 1..*num* ⇒

  This warning indicates that the user forgot to specify both a lower and an upper bound for either an array subscript or the value range for a state-space variable. ASSIST had to assume what the user meant. The assumption, for example, was that "ARRAY[2] OF 5" was supposed to read "ARRAY[1..2] OF 1..5". Since the assumption may not always be correct, the user should specify the full legal syntax and specifically say what is meant. This serious warning message appears when one or more levels of error reporting are requested.

- EMPTY LIST SPECIFIED. ⇒

  This warning indicates that the user followed a left parenthesis by a right parenthesis or a left bracket by a right bracket. It could also indicate that an INPUT statement did not specify any constants to be input, that a SPACE statement did not have any state-space variables, or that a TRANTO did not list any destination expressions. The enclosed text may be in: "[(* 1,2 *)]". This serious warning message appears when one or more levels of error reporting are requested.

- IDENTIFIER TRUNCATED: 28 CHARS MAX ⇒

  This warning indicates that the identifier in question was truncated to 28 characters in length. The arrow ("^") points to the identifier that was too long. This warning message appears when two or more levels of error reporting are requested.

- INPUT LINE TOO LONG. ⇒

  This warning indicates that an input line has been truncated and that part of the information from the end of the line was thrown away. This almost always results in the propagation of at least one error, unless, of course, the truncated text was just part of a comment. This serious warning message appears when one or more levels of error reporting are requested.

- INTERNAL ERROR. PLEASE CONTACT COMPILER SUPPORT: *token* ⇒

  As a warning, this message indicates that ASSIST could not anticipate potential problems which are normally checked. If the standard error file reads "Algorithm to check for TRANTO destination duplication errors flushed in: "qq_parse_space_expr_list"", then ASSIST could not fully check for as many errors as it could have if the algorithm had not been flushed. The generated model is probably correct, but the user should check it more thoroughly. This serious warning message appears when one or more levels of error reporting are requested.

- LITERAL CHARACTER STRING TRUNCATED: *string* ⇒

  This warning indicates that the text of a SURE statement or an INPUT prompt had to be truncated because it was too long. This serious warning message appears when one or more levels of error reporting are requested.

- MISSING TOKEN INSERTED BY PARSER: = ⇒
  This warning indicates that a token was missing. Since it was obvious what it was, it was fixed. This serious warning message appears when one or more levels of error reporting are requested.

- MODEL CONTAINS RECOVERY TRANSITIONS DIRECTLY TO DEATH STATE AND THEREFORE MAY NOT BE SUITED TO TRIMMING: *id* ⇒
  This warning indicates that the user trimmed the model in such a way that the assumptions made when proving the theorem for the trimming bound are no longer valid; therefore, the user may not be able to trust the answer. This serious warning message appears when one or more levels of error reporting are requested.

- MODEL CONTAINS RECOVERY TRANSITIONS DIRECTLY TO PRUNE STATE AND THEREFORE MAY NOT BE SUITED TO TRIMMING: *id* ⇒
  This warning indicates that the user trimmed the model in such a way that the assumptions made when proving the theorem for the trimming bound are no longer valid; therefore, the user may not be able to trust the answer. This serious warning message appears when one or more levels of error reporting are requested.

- NO TRANSITIONS GENERATED USING TRANTO ON LINE: *linenumber* ⇒
  This warning indicates that the TRANTO on the specified line was never used. This warning may signal a logical error in the ASSIST input file; however, the need for this TRANTO statement may have been obviated by pruning. If so, the message can probably be ignored, but the user should check the input file first to make sure that everything is correct. This warning message appears when two or more levels of error reporting are requested.

- NO TRANSITIONS OUT OF A NON-DEATHIF STATE.
  THIS STATE IS THEREFORE IMPLICITLY A DEATH STATE: *statenumber* ⇒
  This warning indicates that the state *statenumber* did not have any transitions leaving it, yet it did not conform to any of the DEATHIF or PRUNEIF conditions. Because there are no transitions leaving it, SURE will assume this state is a death state. This serious warning message appears when one or more levels of error reporting are requested.

- NOT YET IMPLEMENTED. ⇒
  This warning indicates that the user tried to use a future option of a pending version of ASSIST that is nearing completion but still under development and not yet supported. At release of ASSIST 7.0 there are no features that generate this warning. This serious warning message appears when one or more levels of error reporting are requested.

- NUMBER SHOULD BEGIN WITH DIGIT, NOT DECIMAL POINT. ⇒
  This warning indicates that the user began a number with a decimal point. There is currently no place in the ASSIST language where this would be ambiguous, so this warning never appears unless the user requests all warnings. If the syntax ever changes so as to cause this to present a real ambiguity, then the warning will be changed to an error, and users will have to change their input files. The documentation in section 3.1.2 indicates that the decimal point is required to reserve the right to extend the language in the future. This minor warning message appears when three or more levels of error reporting are requested.

- NUMBER STRING TRUNCATED: 27 CHARS MAX ⇒
  This warning indicates that a number string was too long and therefore was truncated. If a precision digit was lost, then the problem is minimal. If an exponent digit was lost, then the problem is serious. This serious warning message appears when one or more levels of error reporting are requested.

- OLD SYNTAX SPECIFIED. PLEASE REPLACE WITH NEW SYNTAX. ⇒
  This warning indicates that the user is using the old syntax for a construct. The old syntax is still legal, hence the warning only appears when users ask for all warnings. The old syntax for the FOR

construct can be ambiguous in the user's mind if that user is using both the old and new syntax in different ASSIST input files. For example, "III = 1,3" for the old syntax includes the number "2" in the range whereas the new syntax "III IN [1,3]" does not include the number "2". This minor warning message appears when three or more levels of error reporting are requested.

- OPTION ONLY VALID ON A UNIX SYSTEM. OPTION IGNORED: *–opt* ⇒
  Certain options, such as **–pipe**, are valid only on UNIX systems. They will not work on VMS. This serious warning message appears when 1 or more levels of error reporting are requested.

- PROGRAM DOES NOT CONTAIN ANY DEATHIF STATEMENTS: *token* ⇒
  This warning indicates that the program did not contain any DEATHIF statements. This warning message appears when 2 or more levels of error reporting are requested.

- TESTS FOR EQUALITY/INEQUALITY OF REALS CAN PRODUCE INCORRECT RESULTS. ⇒
  This warning indicates that the user tried to test for one real "equal" to or "not equal" to another real. The user should use relations such as "<=" or ">=" instead of "==" and "<>". This minor warning message appears when three or more levels of error reporting are requested.

- THE "C_OPTION" STATEMENT SHOULD APPEAR BEFORE ANY OTHER STATEMENTS IN THE INPUT FILE: *token* ⇒
  This warning indicates that a "C_OPTION" statement appeared out of sequence. In most instances, this will cause serious problems. In some instances, this may be acceptable. For example, it is okay to change the maximum number of errors/line at various places in the input file, but increasing memory allocation limits must be done first. This option can be used as the last line of the input file to change the warning/error limits for the rule generation phase. This serious warning message appears when one or more levels of error reporting are requested.

- THIS STATEMENT IS INDEPENDENT OF FOR INDEX: *id* ⇒
  This warning indicates to the user that the specified statement does not depend upon the FOR loop index to the right of the colon. The statement could therefore safely be moved out of the FOR construct for clarity and simplicity. The user should check the statement to confirm that no intended references to the FOR index were omitted. This warning message appears when two or more levels of error reporting are requested.

- VALUE ASSIGNED TWICE IN SAME TRANTO FOR THIS STATE-SPACE VARIABLE: *id* ⇒
  This warning indicates that the state-space variable was adjusted twice in the same TRANTO as in "TRANTO NWP++,NWP-- BY". The net effect of the transition would be to ignore the first adjustment and use only the final one. In the example the "NWP++" would be ignored and the "NWP--" would be used because the decrementing appeared later in the statement than did the incrementing. This serious warning message appears when 1 or more levels of error reporting are requested. In some instances, as in the case of array state-space variables with variable subscripts, the warning is minor and appears when three or more levels of warning are requested.

Appendix D

Debugging ASSIST Input Files

The most cautious user will occasionally mistype something in an ASSIST input file. Usually, generated error messages will give enough detail to make an appropriate correction. For extremely rare instances in which more information is helpful, there are certain *tricks* that one can use to get that information. These are described herein.

D.1. C_OPTION Statement

The command-line option statement can be used at the beginning of an input file to specify special command-line options that must always be used to correctly process the file in question. For example, if **foo.ast** were to begin with

```
C_OPTION W=NONE;
```

The **-w=none** option may be omitted from the command line as illustrated:

```
assist foo
```

The values specified in the **C_OPTION** statement override any values explicitly given on the command line. For example, if

```
C_OPTION W=NONE;
```

is specified as the first line of the ASSIST input file **foo.ast** and if the command line is specified as

```
assist foo -w=all
```

no warnings will be printed because the `C_OPTION W=NONE;` statement overrides the option as specified on the command line.

D.2. Debug Statement

The `DEBUG$` statement can be used to print out internal information to the log file. Internals can be very technical, so not very much will be said about this statement. The source code provides an analysis for advanced users who understand the internals of ASSIST.

There are two forms of the debug statement that may be of use to an intermediate user. These are

```
DEBUG$ EXPAND$;
DEBUG$ NONE$;
```

If placed around a statement containing an erroneous `FUNCTION` or `IMPLICIT` reference, these commands will detail how the macro was expanded and may shed more light on why the context was erroneous. For example

```
DEBUG$ EXPAND$;
IF F(I) THEN
   DEBUG$ NONE$;
   ...
ENDIF;
```


C-2

D.3. Cross-Reference Map

The **-xref** command-line option can be used to produce a cross-reference map of all identifier, block `IF`, and `FOR` references throughout the input file. For example

```
assist foo -xref
  or
C_OPTION XREF ;
```

D.4. Load Map

The **-loadmap** command-line option can be used to produce an extremely technical dump of the data structures and memory layout after the input file has been parsed and before rule generation begins. The load map is written to the log file and might be of limited use to a select few technically oriented users, but "if you ask for it, then you get it" ☺.

Appendix E

Command-Line Options

The ASSIST command line allows the user to specify options. These options control a number of parameters and allow the user more control over how the ASSIST program executes.

Options must be preceded by a slash under VMS, as in

/map

and they must be preceded by a dash under UNIX, as in

-map

Options may be specified in upper or lowercase. The normal UNIX case sensitivity does not apply to the ASSIST command-line options.

Options may also be typed into the input file via `C_OPTION` commands. These commands must precede all other commands, including any other debug commands. For example, the statement

```
C_OPTION LEL=10;
```

in the input file is the same as the following command-line options:

lel=10
or
-lel=10

The following options are available:

- **-c** → Specifies identifier case sensitivity. Use of **-c** is not recommended since SURE is never case sensitive. Case-sensitive state-space variables are safe to use because they are never passed to SURE. Case-sensitive constant names will cause problems because they are passed to SURE. The default is not having case-sensitive identifier names.
- **-pipe** → This option causes the model output to be written to standard output instead of to a model file and is useful if one wishes to pipe the model directly to SURE. This option is valid only under UNIX. An attempt to use it under VMS will cause ASSIST to print an error message. The default is no rerouting of the model file to the standard output file.
- **-map** → This option causes ASSIST to produce a cross-reference map of all the definitions of and references to identifiers and literal values in the program. The map also tells which `ENDFOR` matches which `FOR` and which `ENDIF` matches which `IF`. It also indicates to which `IF` an `ELSE` belong. Although the map is several pages long, it may help the user find misspelled identifiers. Use of the map is recommended during the first few executions of a new input file. The default is no cross-reference map.
- **-xref** → This option is the same as the **-map** option.
- **-loadmap** → This option is used to request a load map of the internal data structures and memory allocation generated during parsing of the input file. The load map information is extremely technical. The option remains in the language for verification purposes and because it is useful in some rare instances. Its use is **not** recommended. Use of **-xref** is recommended instead. The default is not having a load map.
- **-ss** → This option forces ASSIST to print the warning level as part of each warning message. For example, instead of `[WARNING]`, the message will read `[WARNING SEVERITY 3]`. The default is not displaying the warning severity.

- **-we3** → This option forces ASSIST to display three-letter abbreviations in warning and error messages such as `[ERR]` and `[WRN]`. The default is not abbreviating the words ERROR and WARNING.
- **-bat** → This option causes ASSIST to execute in batch mode. In batch mode the command line is echoed to standard error (usually the user's monitor screen). The default is not to echo the command line used to invoke ASSIST.
- **-wid** =*nnn* → This option specifies the maximum line length plus one. The default is 80 characters and results in an effective input line length of 79 characters.
- **-tab** =*nnn* → This option specifies how many spaces are equivalent to a tab character. The default is four spaces per tab.
- **-nest** =*nnn* → Specifies how deeply a space statement can be recursively nested. The default is 16 on most systems (8 on the IBM PC).
- **-rule** =*nnn* → Specifies the maximum number of rules that can be nested inside a single block `IF` or `FOR` construct. The default is 4096 for most systems (1024 on the IBM PC).
- **-pic** =*nnn* → Specifies the maximum number of nodes that can be on the stack when parsing a state-space picture. The number of state-space variables may exceed this number only if the state-space picture is recursively defined. The default is 100.
- **-lel** =*nnn* → Specifies the "line error limit." If the number of errors per line ever exceeds this value, then ASSIST will quit processing the input file immediately after printing one additional and appropriate error message. The default is a maximum of five errors allowed per line.
- **-lwl** =*nnn* → Specifies the "line warning limit." If the number of warnings per line ever exceeds this value, then ASSIST will quit processing the input file immediately after printing an appropriate error message. The default is a maximum of five warnings allowed per line.
- **-el** =*nnn* → Specifies the "error limit." If the cumulative errors ever exceed this value, then ASSIST will quit processing the input file immediately after printing one additional and appropriate error message. The default is a maximum of 40 errors allowed per input file.
- **-wl** =*nnn* → Specifies the "warning limit." If the cumulative number of warnings ever exceeds this value, then ASSIST will quit processing the input file immediately after printing an appropriate error message. The default is a maximum of 40 warnings allowed per input file.
- **-bc** =*nnn* → Specifies the "bucket count" for the rule generation state-hashing algorithm. If rule generation is taking a long time because of identifier hash clashes, this value can be adjusted. The default bucket count is 1009.
- **-bi** =*nnn* → Specifies the "bucket increment." The bucket increment controls how many additional state buckets will be allocated at a time when the system runs out of buckets.
- **-bw** =*nnn* → Specifies the "bucket width" (i.e., the number of states that will fit in a single link of the linked list for each bucket) for the rule generation state-hashing algorithm. If rule generation is taking a long time because of identifier hash clashes, this value can be adjusted. The default bucket width is 5.
- **-lp** =*nnn* → Specifies the number of lines per page on the log file. The default is 58 lines maximum per page on the log file.
- **-i** =*nnn* → Specifies the maximum number of identifiers that can be held in the identifier table. The default is a maximum of 400 unique identifier names in the table for most systems (200 on the IBM PC).
- **-n** =*nnn* → Specifies the maximum number of literal values that can be held in the identifier table. Note that "6.0" and "6.00" are considered as two different entries in the table so that they can be

written to the model file the same way they were typed into the input file. The default is a maximum of 200 unique numerical values in the table for most systems (50 on the IBM PC).

- **-o** =*nnn* → Specifies the maximum number of operands that can be held in the expression operand list while parsing a single statement. The default is 300 on most systems (50 on the IBM PC). The maximum number of infix/postfix operations is a function of this number and is always significantly greater.
- **-e** =*nnn* → Specifies the maximum number of expressions that can be held while parsing a single statement. The default is 300 on most systems (50 on the IBM PC).
- **-p** =*nnn* → Specifies the maximum number of identifiers for a FUNCTION or IMPLICIT or VARIABLE parameter list. The default is 64 on most systems (32 on the IBM PC).
- **-b** =*nnn* → Specifies the maximum number of tokens in the body per FUNCTION, IMPLICIT, or VARIABLE definition. The default is 1024 on most systems (256 on the IBM PC).
- **-w** =*nnn* → Specifies the levels of warnings that will be issued. The higher the number, the more warnings. Levels available are 0 for no warnings through 99 for all warnings. There are currently only three levels defined. The default is two levels of warning reporting. The `W=FEWER` form decreases the level to one less level of warnings. The `W=NONE` form suppresses all warnings. The `W=ALL` form enables all warnings.

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.; and White, Allan L.: *SURE Reliability Analysis: Program and Mathematics.* NASA TP-2764, 1988.

3. Butler, R. W.: *An Abstract Specification Language for Markov Reliability Models.* NASA TM-86423, 1985.

4. Butler, Ricky W.; and Stevenson, Philip H.: *The PAWS and STEM Reliability Analysis Programs.* NASA TM-100572, 1988.

5. Butler, Ricky W.; and Johnson, Sally C.: *Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach.* NASA RP-1348, 1995.

6. Johnson, Sally C.: Reliability Analysis of Large, Complex Systems Using ASSIST. A Collection of Technical Papers—*AIAA/IEEE 8th Digital Avionics Systems Conference,* Part 1, Oct. 1988, pp. 227–234. (Available as AIAA-88-3898.)

7. White, Allan L.; and Palumbo, Daniel L.: State Reduction for Semi-Markov Reliability Models. *Proceedings of the Annual Reliability and Maintainability Symposium,* IEEE, 1990, pp. 280–285.

8. Johnson, S. C.; and Butler, R. W.: A Table-Oriented Interface for Reliability Modeling of Fault-Tolerant Architectures. *Proceedings of the IEEE/AIAA 11th Digital Avionics Systems Conference,* 1992, pp. 149–154.

9. Palumbo, D. L.: Using Failure Modes and Effects Simulation as a Means of Reliability Analysis. *Proceedings of the IEEE/AIAA 11th Digital Avionics Systems Conference,* 1992, pp. 102–107.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1995	Technical Memorandum

4. TITLE AND SUBTITLE
ASSIST User Manual

5. FUNDING NUMBERS
WU 505-64-10-07

6. AUTHOR(S)
Sally C. Johnson and David P. Boerschlein

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17407

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-4592

11. SUPPLEMENTARY NOTES
Johnson: Langley Research Center, Hampton, VA; Boerschlein: Lockheed Engineering & Sciences Company, Hampton, VA.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Semi-Markov models can be used to analyze the reliability of virtually any fault-tolerant system. However, the process of delineating all the states and transitions in a complex system model can be devastatingly tedious and error prone. The Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST) computer program allows the user to describe the semi-Markov model in a high-level language. Instead of listing the individual model states, the user specifies the rules governing the behavior of the system, and these are used to generate the model automatically. A few statements in the abstract language can describe a very large, complex model. Because no assumptions are made about the system being modeled, ASSIST can be used to generate models describing the behavior of any system. The ASSIST program and its input language are described and illustrated by examples.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Reliability analysis; Fault-tolerant systems	101
	16. PRICE CODE: A06

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