NASA Contractor Report **182005**

Reliability Model Generator Specification

G. C. Cohen
Boeing Advanced Systems
Seattle, Washington

C. M. McCann
Boeing Advanced Systems
Seattle, Washington

NASA Contract NAS1-18099
March 1990

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

PREFACE

This report describes the Reliability Model Generator, an analysis tool that produces a reliability model of a candidate system from a structural and functional system configuration. This work has been suported under NASA contract NAS1-10899, Integrated Airframe/Propulsion Control System Architecture (IAPSA II).

The NASA technical monitor for this work is Daniel L. Palumbo, of the NASA Langley Research Center, Hampton Virginia.

The work was accomplished by the Flight Controls Technology organization at Boeing Advanced Systems in Seattle, Washington. Personnel responsible for the work perfomed include -

D. Gangsaas	Responsible Manager
T.M. Richardson	Program Manager
G.C. Cohen	Principal Investigator
C.M. McCann	Flight Controls Technology

Table of Contents

Reliability Model Generator

1. SUMMARY

This report describes the Reliability Model Generator, an analysis tool that produces a reliability model of a candidate system from a structural and functional system configuration. This work has been supported under NASA contract NAS1-10899, Integrated Airframe/Propulsion Control System Architecture (IAPSA II).

This report begins with a brief account of motivation for the tool. This is followed by a description of the tool, detailing the algorithms for each process. Changes to the algorithms since September 1987 are itemized in appendix A. The algorithms themselves are listed in appendix B. Two example traces of the algorithms are listed in appendices C and D.

Currently, a prototype system for the Reliability Model Generator is under development. The resilience of the prototype will be tested on the candidate architecture being designed for the IAPSA II contract, which is based on the tightly synchronized fault-tolerant architecture, the Advanced Information Processing System (AIPS), developed by

The Charles Stark Draper Lab.

2. INTRODUCTION

The use of digital systems and redundancy management schemes to satisfy flight control system requirements of high-performance aircraft has increased both the number of implementation alternatives and the overall system design complexity. Consequently, a comprehensive reliability analysis of each candidate architecture becomes tedious, time consuming, and costly. Evaluation tools do exist that will aid in this analysis process. Given system reliability models (e.g., fault trees, Markov models), these tools will quantify system attributes (i.e., mean time between failures and critical component failure vulnerabilities, etc.) for mission safety, mission success, or for other reliability attributes as deemed necessary.

In order to define the reliability model that serves as input to the evaluation tools, a failure modes effects analysis (FMEA) of the candidate system must be performed manually to determine the effects of component failures on the system. For advanced avionics systems incorporating complex redundancy management schemes, this can involve exploration of system component interrelationships approaching combinatorial explosion. Because of this complexity, an analysis tool called the Reliability Model Generator is proposed that will incorporate failure analysis techniques to generate the reliability model from a functional and structural description of a candidate architecture. This reliability model can then be used by an existing evaluation tool

that solves the model and defines the numeric bounds on system reliability. Figure 1 shows this process.

This report begins, in section 2, with a discussion of reliability analysis techniques used for the IAPSA II project and other attributes of reliability analysis that shape the environment for the Reliability Model Generator which is then described in section 3. Finally, future developmental efforts and enhancements are discussed in section 4.

3. RELIABILITY ANALYSIS

Reliability analysis can be defined as the analysis of events that contribute to the occurrence of undesirable conditions, and the application of probability theory to determine that the likelihood of these undesirable conditions lies within acceptable limits. Undesirable conditions are defined as a nonfulfillment of the system requirements being supported by a candidate architecture (e.g., loss of critical flight control functions). Furthermore, these conditions are a manifestation of component failures propagated through the interrelationship between system components. Therefore, to determine the sequence of component failures that contributes to a particular undesirable condition, an FMEA is performed that traces the effects of component failures according to component interactions. For highly reliable systems, additional functions are incorporated into the architecture for failure detection, isolation, and recovery (FDIR). FMEA must also identify these FDIR mechanisms and analyze their

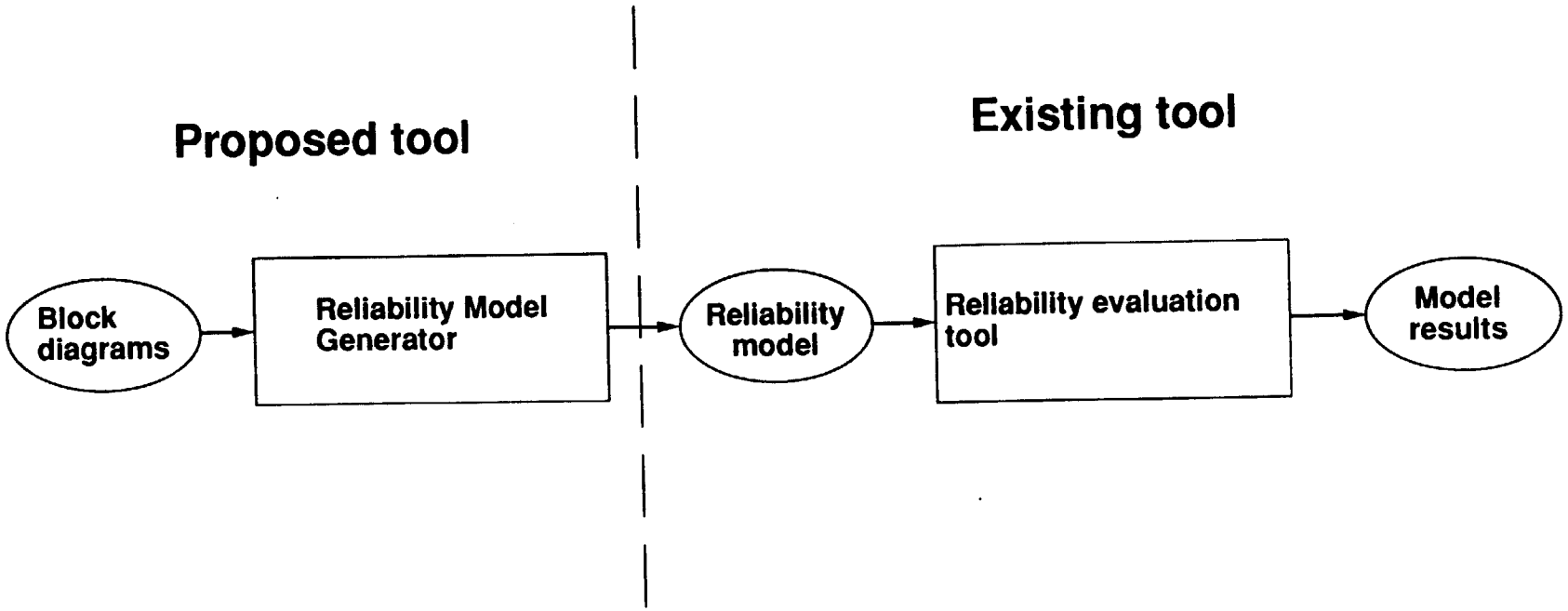


Figure 1. Proposed System

effects on overall system reliability. Another critical aspect of FMEA is concerned with the effects of multiple failures on the system and the effects of nearly simultaneous failures - a particular state of vulnerability in which a second failure may occur before the system can recover from the first failure. These time dependencies contribute to the difficulty of an accurate reliability analysis.

Once an analysis of critical failure modes is complete, a reliability model incorporating these characteristics is defined. The reliability model is then solved by an evaluation tool. The evaluation tool used in IAPSA II is the Semi-Markov Unreliability Range Evaluator (SURE), developed by NASA Langley. A system in SURE is defined as a state space description: the set of all feasible states of the system, given an initial state. State transitions, in SURE, describe the occurrence of faults and fault recovery actions that cause the system to change from one state to another. Given the state space description, including an identification of the initial state and those states that represent an unreliable system, SURE computes the upper and lower bounds on system reliability and provides an enumeration of all system failures. The sequence of component failures that contributed to each system failure is also identified.

An interface to SURE is provided by the Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST), a tool to aid in the specification of the reliability model. Input to ASSIST comprises a state space vector representing

the attributes of the system. The failure modes and FDIR attributes are described to ASSIST as transitions in the form of logical statements. Each transition describes (in terms of the state space vector elements) a logical condition under which a change to the system occurs. The undesirable conditions, called death states, are identified by logical relationships among the state vector elements. From this specification, the SURE model is generated and solved. An example of an ASSIST description is shown in figure 2.

Despite the user-friendly front-end to SURE provided by ASSIST, modeling expertise is needed to efficiently describe the reliability attributes in terms of a state space vector, death conditions, and transitions. In addition, for large system incorporating many components, the SURE state space for the system may culminate in an explosion of states that will require excessive computing resources to solve and inhibit validation of the model.

The Reliability Model Generator aids in analyzing the effects of component failures on other components in the system and outputs a reliability model in ASSIST syntax. The reliability model then can be examined by the user or inputed to ASSIST and SURE to compute the reliability metrics.

In addition to the mechanics of reliability analysis, several environmental attributes merit consideration.

```
SPACE    = ( NGFTP1: 0..1,        (* FTP CHANNEL STATUS          *)
             NPAR11: 0..1,        (* PARTITION INTERFACE STATUS *)
             NGFTP2: 0..1,        (* FTP CHANNEL STATUS          *)
             NPAR12: 0..1,        (* PARTITION INTERFACE STATUS *)
             NPAR22: 0..1,        (* PARTITION INTERFACE STATUS *)
             NGFTP3: 0..1,        (* FTP CHANNEL STATUS          *)
             NPAR13: 0..1,        (* PARTITION INTERFACE STATUS *)
             NPAR23: 0..1,        (* PARTITION INTERFACE STATUS *)
             NGFTP4: 0..1,        (* FTP CHANNEL STATUS          *)
             NPAR24: 0..1);       (* PARTITION INTERFACE STATUS *)

START    = ( 1,1, 1,1,1, 1,1,1, 1,1 );

DEATHIF    NGFTP1 + NGFTP2 + NGFTP3 + NGFTP4 < 2
       OR NPAR11 + NPAR12 + NPAR13     (** SINGLE PARTITION SUCCESS CASE **)
       +  NPAR22 + NPAR23 + NPAR24  < 1;  (** SINGLE PARTITION SUCCESS **)


LAMFTP   = 220.0E-6;          (* FTP CHANNEL FAILURE RATE                *)
                              (*   ---INCLUDES CENTRAL POWER SOURCE---   *)
LAMCOM   =  40.0E-6;          (* FTP NETWORK INTERFACE FAILURE RATE      *)
                              (*   ---INCLUDES ROOT NODE---              *)

IF NGFTP1 > 0 TRANTO NGFTP1 = 0, NPAR11 = 0
              BY LAMFTP;
IF NGFTP2 > 0 TRANTO NGFTP2 = 0, NPAR12 = 0,
              NPAR22 = 0
              BY LAMFTP;
IF NGFTP3 > 0 TRANTO NGFTP3 = 0, NPAR13 = 0,
              NPAR23 = 0
              BY LAMFTP;
IF NGFTP4 > 0 TRANTO NGFTP4 = 0, NPAR24 = 0
              BY LAMFTP;

IF NPAR11 > 0 TRANTO NPAR11 = 0 BY LAMCOM;
IF NPAR12 > 0 TRANTO NPAR12 = 0 BY LAMCOM;
IF NPAR13 > 0 TRANTO NPAR13 = 0 BY LAMCOM;
IF NPAR22 > 0 TRANTO NPAR22 = 0 BY LAMCOM;
IF NPAR23 > 0 TRANTO NPAR23 = 0 BY LAMCOM;
IF NPAR24 > 0 TRANTO NPAR24 = 0 BY LAMCOM;
```

Figure 2. Manually generated ASSIST Reliability Model

Reliability analysis is performed at all phases of the design process. Consequently, models are often built incrementally, starting with limited or cursory knowledge of basic functions and critical failure modes, adding functional information and failure modes as implementation details become available.

At any phase of the analysis, basic units of the architecture are identified, and failure modes postulated for them. These units may correspond to a physical hardware device or may refer to assemblies of units for which composite failure modes are identified. The units have been referred to in literature by various nomenclature including systems and subsystems, elements and subelements, modules and submodules, assemblies and subassemblies, components and subcomponents, structures and substructures, parts, etc. For this discussion, each basic unit of the architecture will be defined as a component. Components may consist of subcomponents, which themselves may be made up of other subcomponents. At some level of analysis, there is an identification of the highest level component and the lowest level subcomponents, and some multilevel hierarchy of subcomponent definition in between.

Failure modes are identified with the lowest level components. At any level of design being analyzed, assumptions are made concerning the level of specification below this level. For example, a multiprocessor system may define each processor as the lowest level component of the system with a

single failure mode.

Analysis at this level makes assumptions about the operation of the subcomponents of each processor. It assumes that no other failures of the processor can be manifested through interaction among a processor's subcomponents. In theory, the more detailed the level of analysis, the more confidence the analyst has in the results. However, as the analysis includes more and more components at increasing levels of detail, the interactions among components through which failures are manifested becomes too numerous to easily analyze.

To manage the analysis complexity, a system may be divided into sets of components. The components in each set are analyzed separately at a detailed level (i.e., several levels of subcomponents), from which critical failure modes are ascertained. Failure modes of subcomponents are combined according to their severity and effects on a higher level component. These failure modes are used to define a model of the component at the higher level. This component then becomes a lowest level component in a new aggregate model (that also accounts for dependencies among the sets). Such incremental analysis allows detailed analysis without an explosion of states. However, care must be taken in this abstraction technique to ensure that an analyst does not overlook failure mode combinations within and between component sets that have a more severe effect on the system than identified. The credibility of the resulting

reliability analysis is only as good as the validity of the assumptions made in the analysis. All assumptions must, therefore, be well understood.

4. RELIABILITY MODEL GENERATOR

This section describes the salient features of the reliability model generator. The overview in figure 3 shows two knowledge bases:

Building Blocks Definition (BBD)

System Definition (SYSD)

and the following software modules:

Model Builder, which is composed of:

1. Reliability Model Aggregation System (RMAS)

2. Local Model Generator (LMG)

Model Reducer/Encoder

Markov Reliability Analysis tool (SURE)

The BBD and SYSD provide a specification of the functional and structural characteristics of the system, respectively, and identify the failure modes for the components. The BBD represents the set of components from which a candidate configuration may be designed. Each component has a specific model that describes its behavior independent of any configuration.

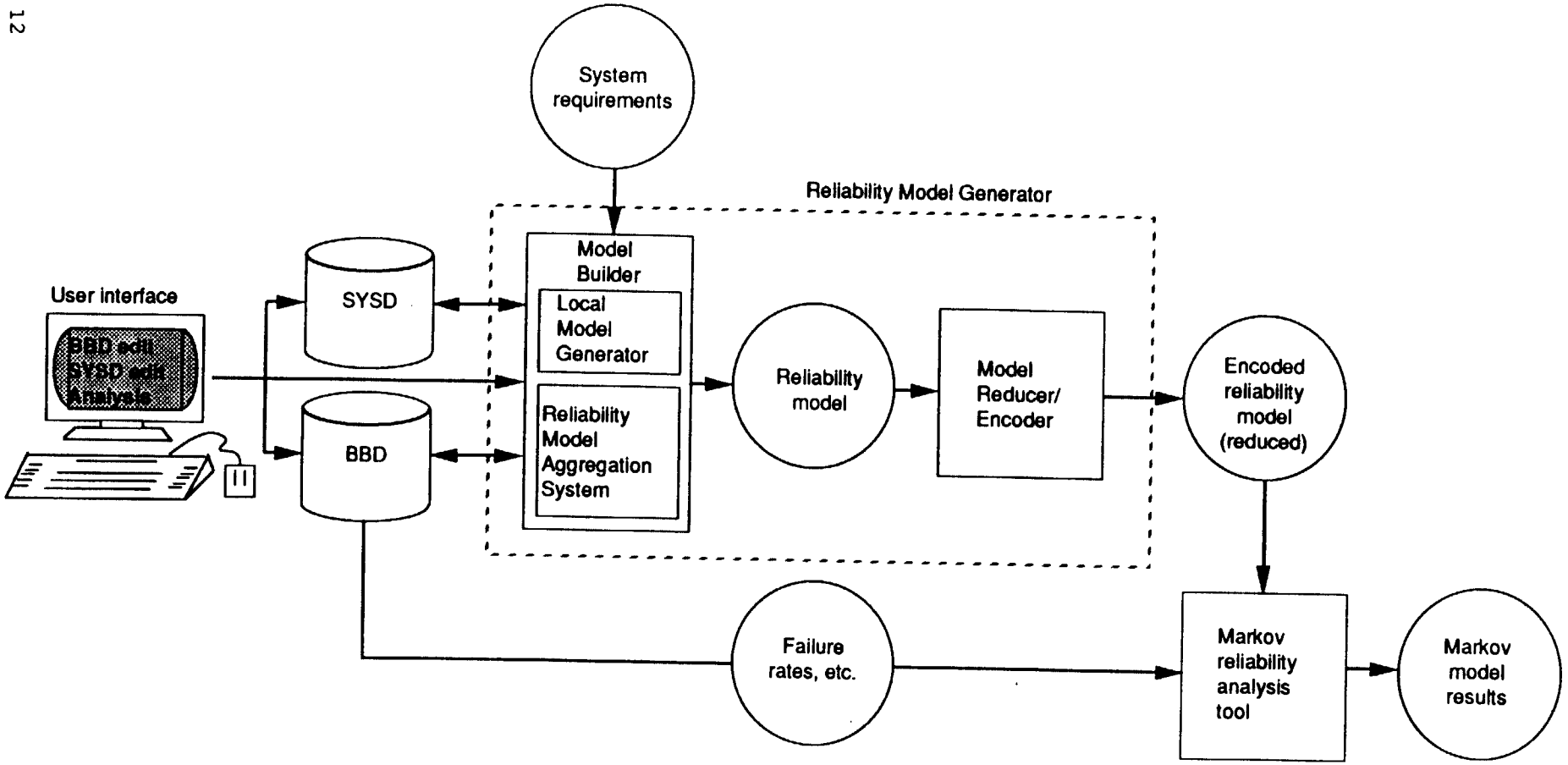


Figure 3. Reliability Model Generator

Once the building block components have been defined in the BBD, the analyst defines a candidate configuration or SYSD. Each component in the SYSD is an instantiation of a component defined in the BBD. The SYSD defines the connections between the component instantiations. The distinction between the SYSD and BBD will be exemplified in the next sections.

Based on the functional behavior and structural description provided by the BBD and SYSD, respectively, and given a global failure condition to analyze, the Model Builder defines a reliability model for the failure condition. The Model Builder (see figure 4) consists of two complementary tools - the Local Model Generator (LMG) and the Reliability Model Aggregation System (RMAS) which may be used separately or in conjunction. The Local Model Generator traces the effects of lowest level component failure modes on other components in the system by following the functional description of the components (defined in the BBD/SYSD). For each of the lowest level components, the Local Model Generator defines a local reliability model. Each local reliability model defines for the component all output effects as a function of the states of the component (i.e. failure modes) and the characteristics of the input to that component (e.g. corrupted and non-corrupted inputs). Component functions, failure modes, and local reliability models will be described further in section 3.1.2.

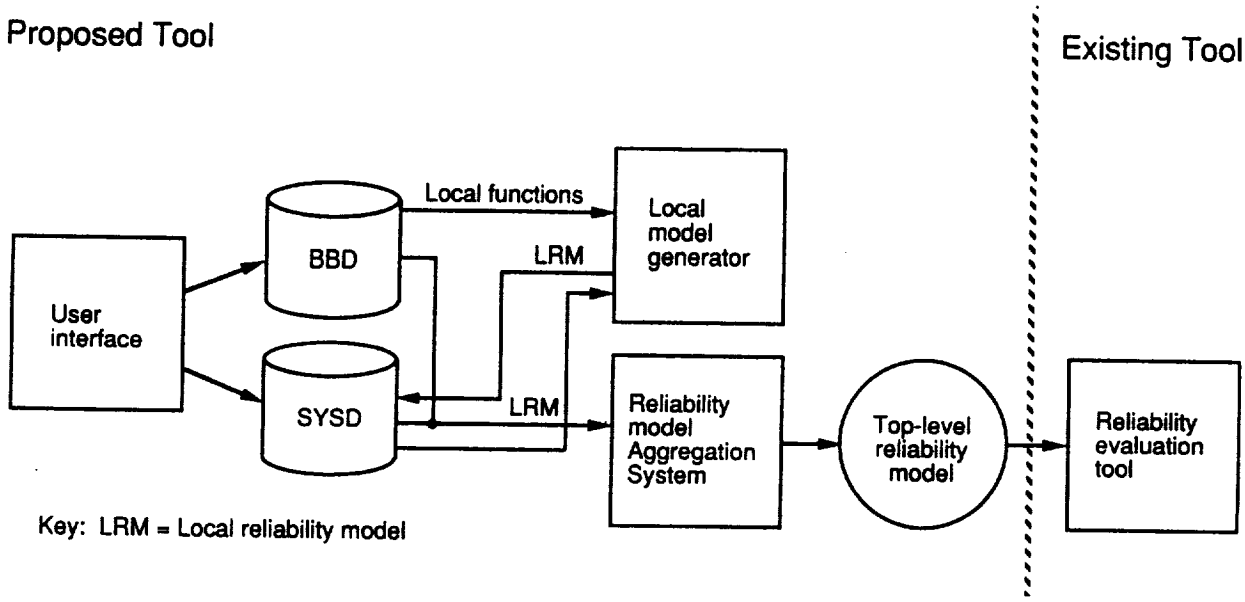


Figure 4. RMAS and LMG

RMAS uses the local reliability models (created by the LMG or entered directly by the user) and knowledge of the interrelationship between components (provided by the BBD and SYSD) to aggregate the local reliability models into a global reliability model for the system. Thus, the Local Model Generator and the Reliability Model Aggregation System together define a model of the system that maps the lowest level failure modes into the highest level unreliable condition.

Once a global reliability model is defined, further reduction techniques are applied by the Model Reducer/Encoder to reduce the model state space and encode the global model into ASSIST syntax from which the SURE model is built. The model is then solved by SURE.

4.1. BUILDING BLOCKS DEFINITION

The BBD represents the set of components from which a candidate architecture may be configured. Each component of the BBD has a specific representation describing its behavior independent of any configuration. The representations define, for each component, its functions, the ways in which the component may fail, and the probability associated with that failure.

BBD components are defined hierarchically, with each level corresponding to a different view of the component. At the top level, a component is defined most generally, and at the lowest level, in the most detail.

Figure 5 illustrates how a computer system might be modeled in the BBD. An interactive user interface allows the user to specify the building blocks graphically as shown on the left in figure 5. This is then mapped into a BBD organized as a hierarchy of components as shown on the right in figure 5. At the highest level, the system is represented by two types of components, the computer and the I/O devices. Only the interrelationship between the computer and the I/O devices is defined. Therefore, the computer is represented as a "box" whose function is to receive information from and output information to I/O devices. To specify the internal function of the computer at the next level in the hierarchy (i.e., the computer's BBD component), the BBD identifies two subcomponents, CPU and memory, and defines their interrelationship within the computer. The third level defines the function of the CPU and the memory elements. Subcomponents of the CPU (i.e., the registers and the ALU) are identified and their interrelationship defined. This hierarchical definition may continue to the most detailed level necessary (e.g., gates or transistors).

The hierarchical definition of components in the BBD corresponds to the way systems are normally characterized--subdividing complex systems into simpler ones. It also allows for flexibility in analyzing systems at all levels of design, thus supporting the iterative nature of reliability analysis portrayed in the

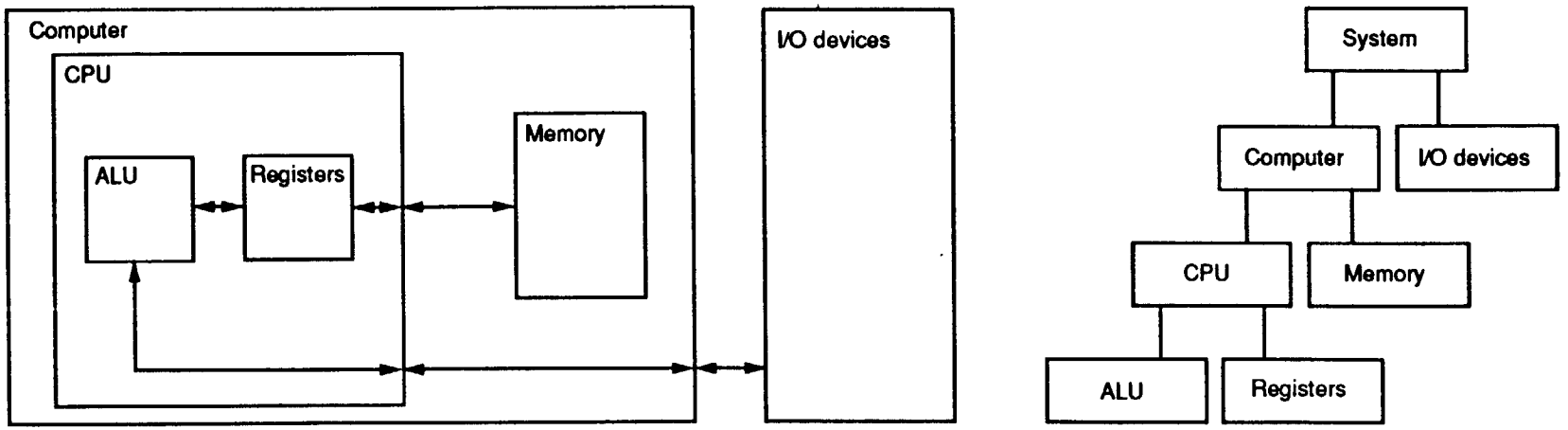


Figure 5. Building Blocks Definition (BBD)

preceding section. At early stages of design when few implementation decisions are known, a high-level view of the system may be defined and analysis of critical failure modes performed based on this model. (Failure modes are discussed in the next section.) When further design details are known and needed in the reliability analysis, subcomponents are defined to expand the functional description, and failure modes are modeled at a more detailed level. As will be further illustrated, this top-down structuring of component functional requirements also allows the tool to trace the role that a lower level component plays in a top-level unreliable condition.

4.1.1. TOP AND INTERMEDIATE LEVEL BBD SPECIFICATION

The highest and intermediate level BBD component descriptions define the function for that level by identifying the subcomponents involved in that function and describing the way in which the subcomponents interact.

The functional flow between subcomponents is defined using the ";" symbol to indicate sequence and the "@" symbol to indicate parallelism. Parallelism amoung redundant components may also be specified by using the FA (for all) universal quantifier. Each subcomponent is identified in the functional flow by its name, and following the name is a specification of its inputs and outputs (separated by a "::"). The function performed by each subcomponent is not

specified in the BBD for component A, but rather is specified in a separate BBD component module for each subcomponent at the next lower level in the BBD hierarchy.

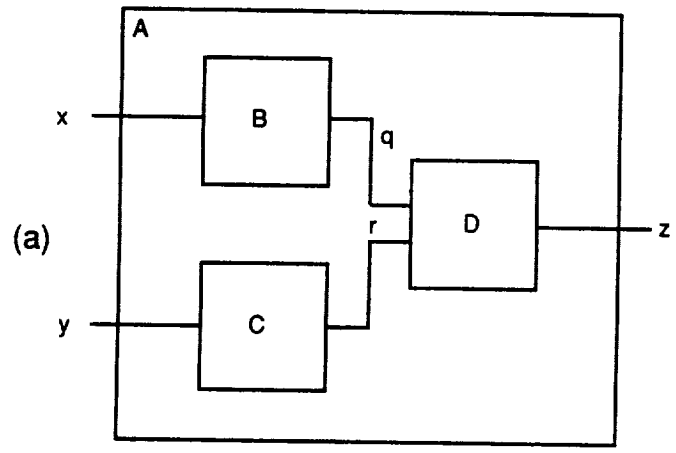
Figure 6a shows a block diagram of a simple system and figure 6b shows an example of the BBD specification for component A. Component A is composed of three subcomponents, B, C, and D. Subcomponents B and C execute in parallel on inputs x and y, respectively. The outputs from B and C (q and r, respectively) are sent to subcomponent D which outputs the final value, z.

The current prototype under development uses a graphical interface to create components and subcomponents in the BBD and to specify the functional flow between subcomponents within a parent component. With this interface, the user need only connect lines between components on a screen (as in figure 6a), and the system creates the internal specification shown in figure 6b. The user may also modify the internal representation.

4.1.2. LOWEST LEVEL COMPONENT BBD SPECIFICATION

The lowest level component description defines the component function and any failure modes that are to be analyzed.

Before defining the functional specification for the lowest level components, however, it is necessary to introduce the concept of input and output characteristics which



(b)

```
A : : @ ( B : : input x; output q;
          C : : input y; output r;);
          D : : input q, r; output z;
```

Figure 6. Intermediate Level Function Specification

is a central theme in the Model Builder processes.

4.1.2.1. INPUT/OUTPUT CHARACTERISTICS

Input/output characteristics are used to represent the effects of component failure modes on information flowing to other components in the system. The motivation for the introduction of input and output characteristics is this: A component's failure modes affect its outputs which are then progagated to other components. Rarely does the effect of a failure mode identify the exact value of the output for a component that fails. Rather, only certain characteristics of the output can be defined. For example, total component failure is a commonly modeled failure mode in which the expected component output is incorrect or corrupted in some manner. The exact value outputted is not so evident in the analysis as the fact that the value is not the expected value. The output of this component is propagated to another component, whose behavior is affected by the presence of this input aberration. The Model Builder processes must therefore "reason" about effects of corrupted input data on components, regardless of the value inputted.

To address this, qualitative characteristics of inputs and outputs--not values--are propagated. Currently, three characteristics for information are used to define the effects of failure modes on interactions between components:

- GOOD (y:g) characterizes a value (for variable y) as that expected under normal (not failed) conditions.

- BAD (y:b) characterizes a value as corrupted in some unspecified manner (failed) whereby the value is not the expected value.

- NIL (y:n) characterizes a variable that is undefined or whose value was not received on time.

For example, a failure mode effect in which the component's output y is corrupted is represented as y:b.

This characterization of inputs and outputs is sufficient to describe the effects of most failure modes. However, there are some instances in which additional information is needed.

For example, consider a "threshold analyzer" component that outputs the value it receives if that value is within some threshold limit, and outputs an error signal otherwise. The reaction of this component to an erroneous input is dependent on whether or not the corrupted input lies within the threshold limits. However, given only that the input is corrupted in some manner (e.g., y:b), the exact value is not known. Reliability analysts, under these circumstances, estimate the likelihood of each type of corruption based on any one or more of the following assumptions:

1. Assume the worst effect so that the overall model is conservative.

2. Use measurement data for the failure mode that caused the erroneous output to:

a. Model each possibility, adjusting the failure rates of the possible effects by their likelihood of occurrence.

b. Model the most likely possibility and ignore extremely unlikely ones. This may not be conservative if the disregarded condition causes a worse effect.

The criteria for selection of an appropriate approach are dependent on the failure mode that caused the effect and therefore cannot be determined by the tool. To address this, the input and output characteristics for components may be specified non-deterministically such that multiple effects of a failure mode are modeled, and a percentage, representing "likelihood of occurrence," may be associated with each distinguishing effect. Note that although the user may recognize the existence of two possible output characteristics for a given input characteristic combination, statistical data on such occurrences may not be available. Subsequently, assigning a percentage occurrence with an input characteristic makes an assumption on the failure mode characteristics that may be arbitrary. Therefore, the use of non-deterministic models is discouraged.

As another example, consider the error signal generated by the threshold analyzer. Suppose the input is not corrupted and is within the tolerance of the threshold. A signal whose value is "no error" would be generated. The

characteristics of the error signal would be GOOD (i.e., not corrupted). Suppose the input is corrupted and not within the tolerance of the threshold. A signal whose value is "error" is generated to another component. The characteristic of the error signal is still GOOD (i.e., not corrupted). However, the component receiving the error signal needs to distinguish a "GOOD" signal to one that indicates error from a "GOOD" signal that indicates no error in order to determine its course of action. Therefore, propagating only a "GOOD" characteristic for the error signal is not sufficient to analyze the effect of the error signal that resulted from a detection of a failure (threshold violation). The actual "value" of the signal must be propagated in addition to its characteristic. The propagation of variable values is further illustrated for the example trace in appendix D.

To summarize, characteristics of GOOD, BAD, and NIL are used to represent the effects of failure modes on interactions between components. In addition, likelihood of occurrence may be associated with an input characteristic, or the value propagated may be specified with the input characteristic in order to provide compatibility of input and output characteristics between components.

4.1.2.2. FUNCTIONAL DEFINITION

Having defined input and output characteristics, the following are minimum requirements for a functional specification of the lowest level BBD components:

1. Completeness: The specification syntax should be sufficient to model the component functionality. The relationship between the information that is input to the component and the information that is produced by the component should be defined by the syntax so that the effects of inputs (that have been influenced by other component errors) on the component function may be analyzed.

2. Consistency: The functional specification should be concise and unambiguous.

3. Clarity: The analyst should be able to understand the functional syntax.

4. Flexibility: The analyst should be able to specify the component function in the most natural way. Also, when the component functional definition is not well defined, (as in early stages of design analysis), the functional syntax should not force a definition of the implementation.

Lowest Level Functional Syntax

With the above minimum requirements in mind, the BBD components at the lowest level are defined as follows. Each component function will be defined as a series of sentences separated by a ";". Each sentence described a separate action of the function. Within each sentence, one or more clauses are defined. Each clause has two parts separated by

a "|". The right part contains a set of conditions which must hold true for the left part to be evaluated. The left part may be a variable, an exact value or a characteristic which is considered to be the output of the sentence if the condition on the right part holds true. The output of whichever clause holds true is assigned to the variable on the left side of the "=" sign. For example:

```
[1]  y = x | x > z
           z | x <= z;
```

This sentence contains two clauses and states that the variable y will equal x if x > z and will be z if x <= z.

In defining sentences, however, the user must ensure that the component is completely defined on all input combinations, and that clauses within a sentence do not have overlapping conditions. For example, the function:

```
output y    1 | x = 3
            2 | x > 0;
```

must be defined as:

```
output y    1 | x = 3
            2 | x > 0 and ^(x = 3);   {where ^ is NOT}
```

Input/Output Variables.

Input variables (information received from another component) and output variables (any information that can be

seen by another component) are explicitely defined:

```
[2]  Ex:  INPUT x;
          INPUT z;
          OUTPUT y =  x | x > z
                      z | x <= z;
```

This states that the variables x and z are received from other components. The output variable, y, contains information that is sent to other components.

Function Macros

In addition to variables, functions may be specified to the left of the | sign. For example:

```
[3]  Ex:  INPUT x;
          INPUT z;
          OUTPUT y =  (+ x z) | x > z
                      (- z x) | x <= z;
```

This states that the output y will be the sum of x and z if x > z and the difference of z and x otherwise. In order to provide a more user-friendly functional specification, Macros can be defined once and, thereafter, used by the analyst. For example, instead of function [1] above, a macro MAX may be defined:

```
[4]       MAX(x y) --> x | x > y
                       y | x <= y;
```

The "-->" symbol indicates that the MAX function is equivalent to the sentence to the right of "-->". With this macro definition, function [2] could be specified by the user as follows:

```
[5]  Ex:  INPUT x;
          INPUT z;
          OUTPUT y = MAX(x z);
```

However, internally, the representation of the function would remain as in [2].

At this time, the proposed Model Builder contains macros for the following functions:

<

logical comparators

#(<set><cond>)

number of elements satisfying condition

ALL(<set><cond>)

boolean that determines if all elements of set satisfy condition

Nil vs. Non-nil Function Categories

The function for the lowest level components is defined in order to be able to trace the effects of input characteristics through the function and define output characteristics. These output characteristics then become input characteristics for other components. Most functions define

output value as a result of input values. However, for this reliability analysis, functions must define the output characteristics as a result of input characteristics. For example, for an adder, the function "+" is defined on integer inputs. However, the inputs are {g,b,n}. Rules must define output characteristics of {g,b,n} for all possible input characteristics, {g,b,n}.

Two rules for this translation are straightforward:

1. If all operands for a function are "g", then the output of the function is "g".

2. If one or more inputs to a function are "b", and all other inputs to the function are "g", then it can be assumed that the output of the function is "b".

However, what is the output of a function if one input is "n" and another input is "b"? Functions can be categorized into two groups according to this situation:

1. Nil sensitive (NS) operations: the output value of the function is 'sensitive' to the absence of non-existent inputs; that is, if any input is "n", the output is "n".

   Ex. Model mathematical functions (e.g., +,-, etc.) as producing no output if all its operands are not available.

   The rules for defining output characteristics for nil sensitive functions are:

OUTPUT "g" IF all inputs are "g"
"n" IF any input is "n"
"b" IF any input is "b" and no input is "n".

2. Non-nil sensitive (NNS) operations: the output value is not sensitive to nil input values such that any "n" inputs are ignored in the calculation of the output.

Ex. A majority function may ignore nil or non-voting inputs and determine the majority based on the available inputs

The rules for defining output characteristics for non-nil sensitive functions are:

OUTPUT "g" IF all inputs are "g"
"n" IF all inputs are "n"
"b" IF any input is "b"

All mathematical functions, such as +, -, etc. are assumed to be nil sensitive functions. Therefore, a sentence containing these operations will be internally translated into NS operations. For example, the function in [3] would be:

```
[6]  Ex:  INPUT x;
          INPUT z;
          OUTPUT y =  NS(x z)  | x > z
                      NS(x z)  | x <= z;
```

If preferred, the user may specify functions using the NS and NNS functions directly. This type of specification is ideal at early stages of design when functional implementation details are not known. However, the user may prefer to specify the function itself as in [3].

Non-Procedural Functional Specifications

For some components, it is sufficient to define the function in the procedural manner described above. However, for other components, the procedural specification is not as easily defined; nor is it necessary to define it in such a manner.

For example, in describing the function of a VOTER component the user may wish to specify that the voter outputs the majority of the inputs. However, at a high level of design, it may not be known what implementation is involved in the computation of the majority. Further, if defined in a procedural format, the specification results in a nested looping structure with variables for counting the number of occurrences of each input value. It may not be important for the analysis that the means of obtaining the majority is defined; rather, it is only important that the definition of a majority be 'understood' by the system.

In order to add flexibility to account for non-procedural functional specifications, an enhancement to the specification allows the use of universal and existential quantifiers. These quantifiers specify conditions upon

which the output is defined.

As an example of this, the VOTER function could be defined as:

```
[7]  OUTPUT y:nil | ALL(x(i)):nil
                y = t | FA x(i):(^nil) ::                    #(x(i) =
     t) > #(x(i) <> t)
```

This function states that the output, y, will be nil (i.e. no output) if all inputs are nil, and the output, y, will be the value of t if for all inputs not equal to nil, the number of inputs equal to t will be greater than the number of inputs not equal to t. Here, "FA" is used to denote the universal quantifier, and "::" is used to separate the quantifiers from the other conditions in the clause.

The procedural specification differs from the non-procedural specification by the presence of quantifiers in non-procedural clauses. The universal quantifier specifies an attribute that is applicable to a set. The set usually represents redundant or replicated variables. Although similiar, the universal quantifier is not equivalent to the ALL({cond}) predicate. For example, the voter specification in [8] redefined as:

```
[8]        output y:nil | ALL(x(i)):nil
           output y = t | #(x(i) = t) > #(x(i) = z)
                              AND ALL(x(i):^n) AND ALL(z <> t);
```

is not defined on all inputs (e.g., when some x(i):n), but not ALL(x(i):n)). In other words, a simple predicate is "checked" to verify the validity of the clause, and a quantifier alters the inputs so that the clause is valid. For this reason, the quantifier is eliminated from the final OCD whereas all predicates remain in the final model. Analogously, the existential quantifier is not equivalent to a AT_LEAST_ONE(<cond>) predicate.

As another example of the use of quantifiers, consider a type of voter that outputs a plurality of the inputs. In other words, the value outputted may not be the clear majority, but there may be a greater number of these values than any other value inputted. This function would be specified as:

```
[9]  OUTPUT y:nil | ALL(x(i)):nil
                y = t | FA z <> t, x(i):(^nil)
                        :: #(x(i) = t) > #(x(i) = z)
```

This function states that the output, y, will be nil (i.e. no output) if all inputs are nil, and the output will be the value of t if for all values z not equal to t and for all inputs not equal to nil, the number of inputs equal to t will be greater than the number of inputs equal to z. (This voter function is specified for the voter component in the example traces in the appendices C and D.)

In order to make the specification more straightforward for the user, macro definition for majority may be defined,

and the user need only specify:

```
OUTPUT y = MAJ(x(i));
 OR
OUTPUT y = PLU(x(i));
```

for functions in [8] and [9] respectively.

4.1.2.3. FAILURE MODES

Failure modes are defined in the BBD as a change to or an aberration of the component function. Thus at the lowest level, there is a definition of the component function under normal operation and a definition of the function or change to the function for each component failure mode. Most failure modes are defined by a change to the outputs produced by the component function rather than a change to the function itself.

For example, a component X may have a failure mode in which any outputs are corrupted regardless of the inputs. This failure mode would be defined by specifying a component state, X_BAD, whose function is simply to output corrupted data:

```
[10] OUTPUT y:b IF X_BAD   (for output y).
```

For all failure modes, transitions are defined from a non-failed component state. These transitions become part of the local reliability model for the component. A transition is defined for the X_BAD failure mode as follows:

[11] IF X_NOF TRANTO X_BAD by <failure rate>

which states that if component X is not failed (in state X_NOF), then it may enter a failed state X_BAD according to some probability of occurrence of the failure mode. Since the Reliability Model Generator is not concerned with the numerical rate associated with the failure mode, hereafter, the failure rate will be eliminated from the transition.

It is not necessary to specify the system below the level at which failure modes are defined, since only the effects of component failure modes on the system are of interest. Conversely, if the reliability of a system is to be analyzed given a set of failure modes, the system components must be defined at least to the level at which failure modes are identified.

Example BBD

Figure 7 shows the system building block diagram used in trace 1 (of appendix C), and figure 8 shows the BBD components. The first component (figure 8a) is the root or top level component which defines the system inputs and outputs (x and y respectively). Two subcomponents and their relationship are defined at this level. Component A inputs x and outputs q. Component B inputs q and outputs y. The internal function of components A and B is not defined at this level; rather, a separate BBD component details this. For component A (figure 8b), two more subcomponents, P and VOTER are identified. Component P is specified as a

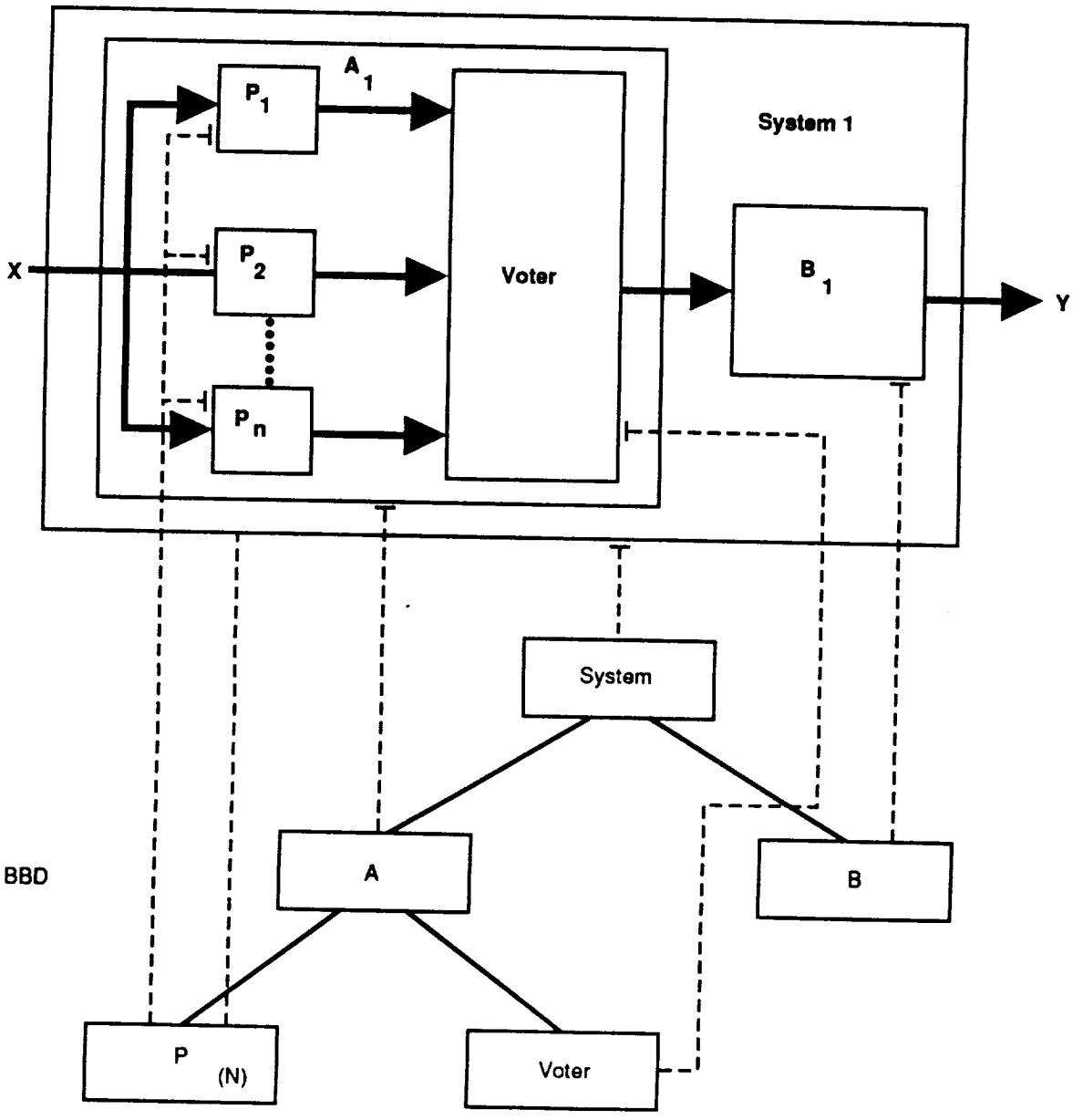


Figure 7. Voted Redundant Processor Example

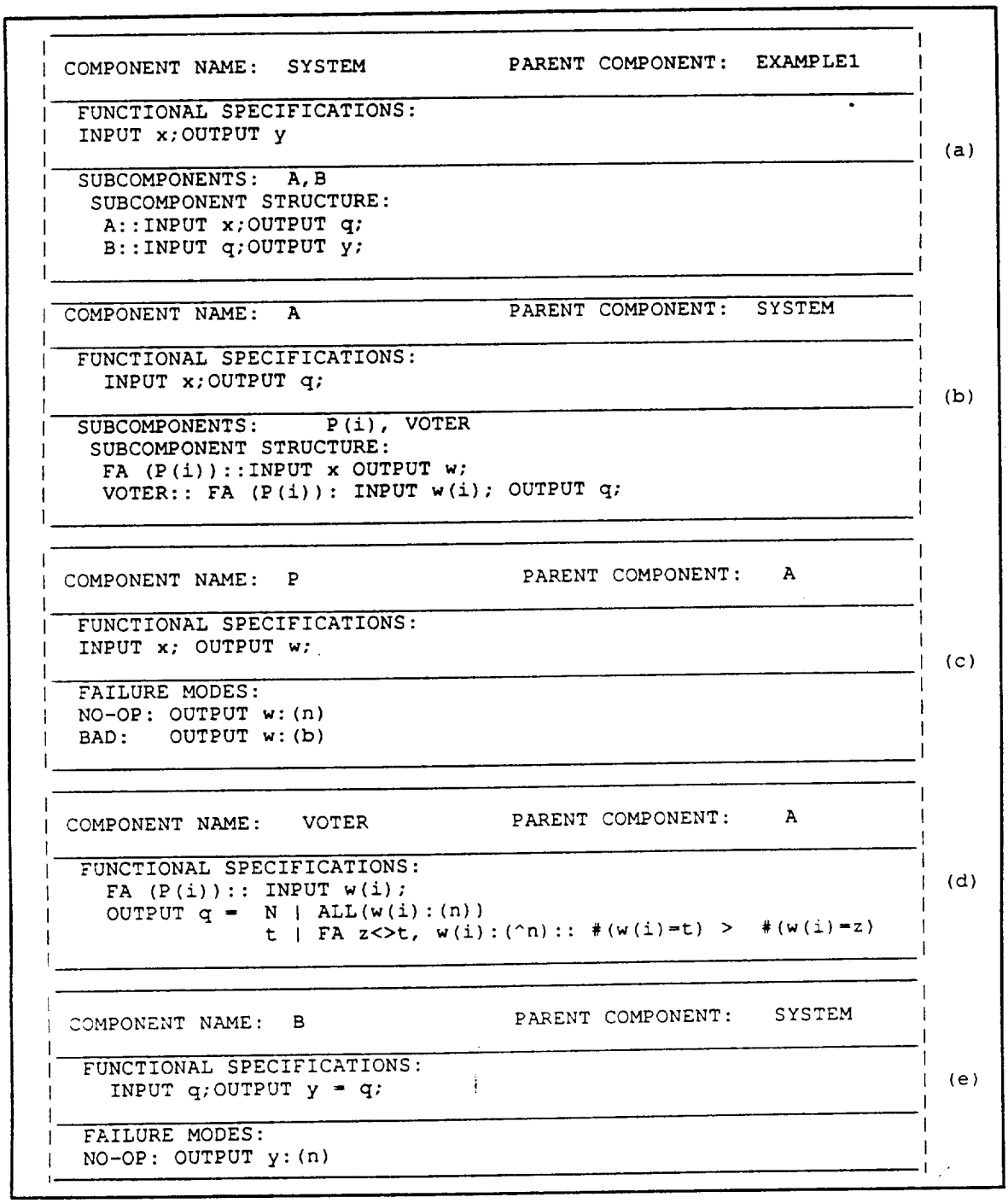


Figure 8. BBD for Example 1

redundant component (denoted by the subscript for P). The substructure for A:

```
FA (P(i))::INPUT x; OUTPUT w;
VOTER:: FA (P(i)): INPUT w(i); OUTPUT q;
```

specifies that all invocations of component P receive the same input, x, and output a variable w. The voter component receives all w(i) from the redundant components, P, and outputs a value q, which is identified as the output of the component A also. Note that the specification of components P and Voter do not specify more than their input and outputs at this level.

Component P is specified in figure 8c. Since P is a component defined at the lowest level in the BBD, two failure modes are identified. The first failure mode is called NO-OP and its specifies that when P fails in this manner, the output w is "n". The second failure mode, called BAD, specifies that the output w is "b" regardless of the inputs.

The voter component is specifed in figure 8d. The function defined is equivalent to the one in [9]. There are no failure modes represented for the voter, even though the voter is a lowest level component.

Finally, component B is specified in figure 8e. A single failure mode, NO-OP, is specified. This failure mode is identical to the NO-OP failure mode for P.

4.1.3. LOCAL RELIABILITY MODELS

Although the local reliability models are not considered part of the BBD, it is important to distinguish their specification and use from that of the lowest level functional definition previously defined. Consider the voter component used in the traces in appendix C and D. The function for the voter is defined as in [8]. The local reliability model for the voter component, defines for all possible output characteristics, the combinations of input characteristics and voter failure modes that cause the output characteristic. If no failure modes for the voter are modeled, its local reliability model is:

```
[12]  OUTPUT y:n IF ALL(x(i):n)
      OUTPUT y:g IF #(x(i):g) > #(x(i):b)
      OUTPUT y:b IF #(x(i):b) > #(x(i):g)
```

If failures are modeled for the voter, then the local reliability model includes transitions for these failure modes (see [11]). Thus a local reliability model contains a set of output characteristic definitions (OCD) for each possible output characteristic, and a set of transitions for each failure mode. Local reliability models must be defined for all lowest level components in the BBD before an aggregate model of the entire SYSD is created by RMAS. Either the user defines these models explicitely, or the LMG is invoked to analyze the functional definitions as in figure 8, trace the effects of component failure modes on other components in the system, and from this trace define the local

reliability model for the components (as in [12]).

Earlier designs of the Reliability Model Generator did not distinguish the Local Model Generator from the Reliability Model Aggregation System. Rather the global model was generated coincident to the failure modes effects analysis (FMEA). However, the current design advocates a separation of these functions for several reasons:

1. Often, the system is decomposed into subcomponents such that the reliability models for the lowest level components of the system are defined by the user directly, rather than requiring the user to specify a component function from which the Local Model Generator creates a local reliability model. If the user specifies for the lowest level components the local reliability models, then the Reliability Model Aggregation System can use these models directly to create the aggregate model for the system.

2. However, in order to define the local reliability models for the system, the user must know all possible abberations (characteristics) of inputs that could possibly affect the components function. This requires an apriori Failure Modes Effects Analysis (FMEA) of the effects of component failures on other components in the system. One of the primary motivations for this tool was to support the FMEA process. Therefore, an automated trace of failure effects (e.g., the LMG)

would be beneficial towards this end.

3. Further, for complex components involving many inputs, many outputs, and many failure modes, a well defined, unambiguous reliability model is difficult to specify.

4. The Local Model Generator can be used to check the accuracy of the reliability models created by the user.

5. The user can easily modify local reliability models directly without having to recreate the entire FMEA analysis of the Local Model Generator.

6. In defining a local reliability model (whether manually or via the LMG), assumptions are made regarding the data characteristics as a result of failure modes. These assumptions (discussed in section 3.4) are interactively verified by the user in the LMG. Therefore, the Local Model Generator is not a tool that is not used without interaction with the user. However, its trace of failure modes could be more thorough than performed by an analyst manually.

The first phase of the prototype implements only the RMAS. With this in mind, the reliability Model Aggregation System is described in section 3.3 followed by the discussion of the Local Model Generator in section 3.4.

4.2. SYSTEM DESCRIPTION

Once the building block components have been defined, the analyst defines a candidate configuration or

system description. Each component in the SYSD is an instantiation of a component defined in the BBD. There may be several instantiations of a single BBD component in the SYSD. The SYSD delineates the scope of the system to be analyzed, such that components in the BBD not instantiated in the SYSD are not included in the analysis. This allows system subsets to be analyzed separately, if required. When a component is selected for instantiation, all subcomponents for that component are instantiated with it to the level at which failure modes are represented in the BBD. This defines the lowest level of abstraction selected for analysis. The top-level unreliable condition to be analyzed is defined with respect to the highest level component of the SYSD. By changing the highest level of the SYSD or by changing the level at which failure modes are defined, a system may be modeled at varying levels of detail without altering the structure of the BBD. This permits easy modification for critical failure mode analysis. Figure 9 shows three possible configuration alternatives for analyzing a simple computer system such as that shown in figure 5.

The prototype currently under development will use a graphical interface to specify instantiation of BBD components into the SYSD and will allow the user to graphically select the connections between components (e.g., mouse and menu).

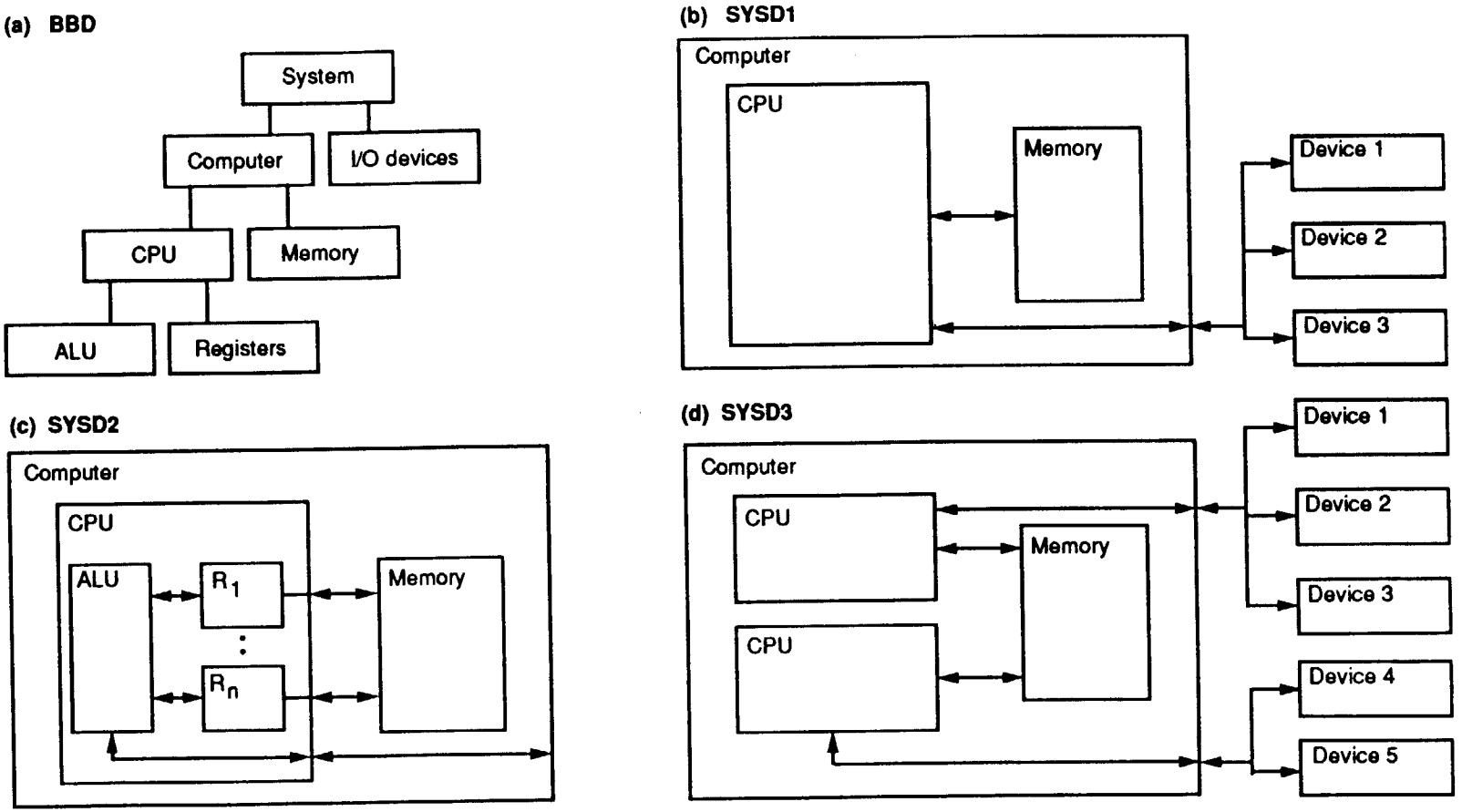


Figure 9. System Definition (SYSD)

BBD/SYSD distinction

The BBD/SYSD distinction allows a functional specification of components independent of their roles in a system. Generally, the BBD and SYSD enforce separation of function and structure so that alternative configurations can be analyzed simply by altering the SYSD. A component's definition may, however, include structural constraints with respect to other components that are common to all instantiations of that component. For example, a multiprocessor may have ports that are always intended to connect to I/O devices. By allowing these constraints to be specified in the BBD description of the component, an instantiation of the component in the SYSD could enforce these constraints, thereby disallowing improper configurations.

4.3. RELIABILITY MODEL AGGREGATION SYSTEM

The Reliability Model Aggregation System (RMAS) uses the local reliability models (created by the LMG or entered directly by the user), defined for the lowest level components of the SYSD to be analyzed, and knowledge of the interrelationship between components (provided by the BBD and SYSD) to aggregate the local reliability models into a global reliability model for the system.

Define the problem instance for the RMAS:

Given a SYSD, a supporting BBD with local reliability

models defined at the lowest level components, and an unreliable condition, define a model of the failure modes that contribute to the unreliable condition.

Recall that the SYSD defines the highest and lowest level view of the system to be analyzed, and at the lowest level, local reliability models define the effects of the component failure modes and input characteristics on the output characteristics of the components. Whether these local reliability models are defined by the user or are generated by the Local Model Generator is irrelevant at this time. The unreliable condition is a definition of the state of the component at the highest level that constitutes an unreliable system. Normally, this is defined as an undesirable output of the function defined at the top level. Therefore, the goal of the analysis is to define a global reliability model of a highest level unreliable condition from the lowest level local reliability models.

Figure 10 shows a hierarchy of Reliability Model Aggregation System modules (RMAS modules) in which each module corresponds to a separate component abstraction (defined by the BBD/SYSD), beginning at the root, or highest level of component description. The purpose of each RMAS module is to define a reliability model for that component level that comprises an aggregate of the lower level modules.

Initially, a RMAS module is instantiated for the highest level component. Given the unreliable condition

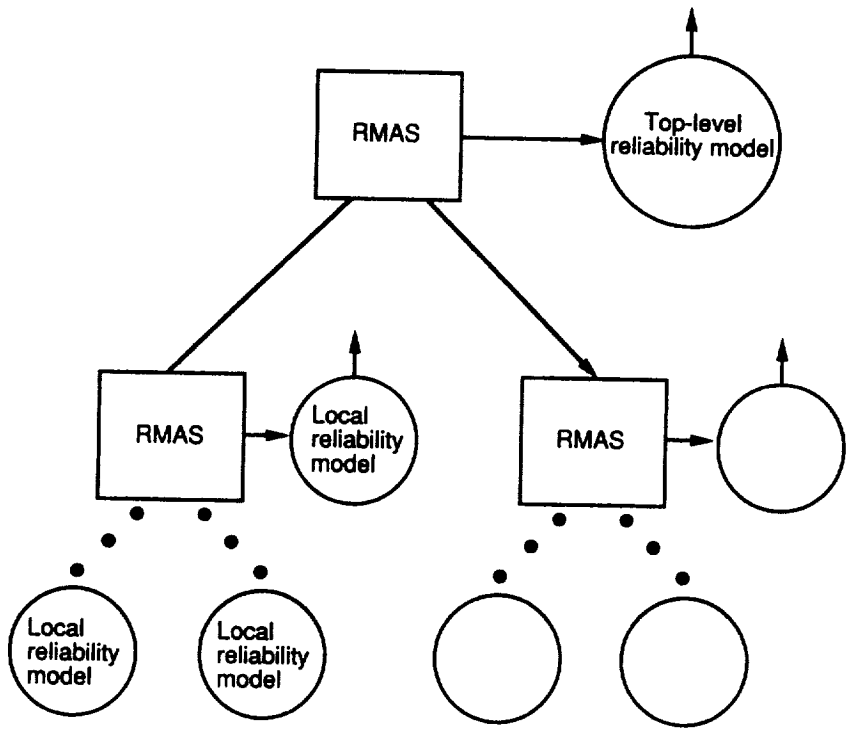


Figure 10. RMAS

specified in terms of an undesirable output defined at this level, RMAS identifies the immediate subcomponents. If the subcomponent is an intermediate level component also, a second level RMAS module is instantiated for the subcomponent. Its subcomponents are identified, and for each successive intermediate level component abstraction, a separate RMAS module is instantiated according to the subcomponent interdependencies defined in the BBD/SYSD for that component level.

If the subcomponent is a lowest level component, a local reliability model is defined which identifies, for each output characteristic, conditions such as: (1) component failure modes on component outputs and (2) erroneous input characteristics, that contributes to the output characteristic.

Each such condition is analyzed by the parent component RMAS module. For failure modes (1), transitions are defined explicitly in the local reliability model for the subcomponent that failed. Each input characteristic (2) corresponds to an output characteristic of some other subcomponent that interacted with the subcomponent. For example, in figure 11, the input characteristic for component C corresponds to the output characteristic for component B. The RMAS for the parent component, therefore, must investigate the model for component B to find transitions that contributed to its output characteristic which, in turn, served as an input characteristic to component C. If component B

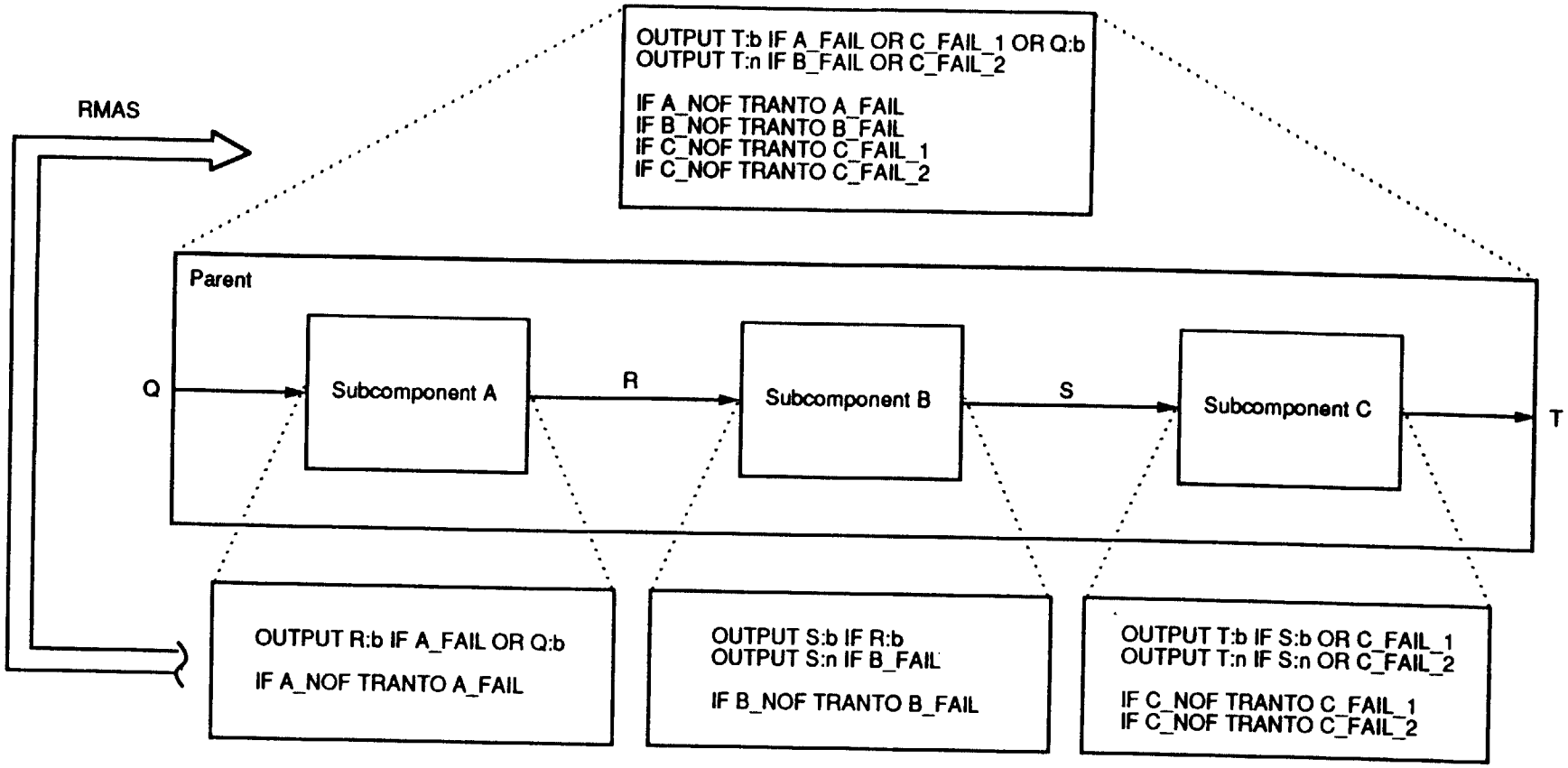


Figure 11. Mechanization of Model RMAS

is an intermediate level component, then, as stated before, a RMAS module is invoked for it (at the third level in the hierarchy). This RMAS module must create a reliability model of component B that defines the states and input characteristics that contribute to B's output characteristic being analyzed by the parent component.

If component B is a lowest level component, then its reliability model is interrogated directly. Component B's output characteristic may be dependent on its state and its input characteristics, etc. At each iteration of this back-tracing, transitions are defined for component state changes, and analysis of input conditions is deferred to the component from which the input was generated. This back-tracing continues until:

1. the output characteristic being analyzed is not derived from an input characteristic,

2. the input of the parent component is reached, or

3. a cycle is detected.

Upon reaching one of these conditions, the "chain of conditions" (i.e., input/output characteristics and failure modes) found among subcomponents are aggregated into a single, local reliability model for the parent component. The failure mode transitions found may be changed to reflect changes to the parent component rather than simply the outputs local to the subcomponent. Further, failure recovery

transitions involving multiple subcomponents (and therefore, not represented in any single subcomponent model) may be detected. This model is returned to the next higher level where the tracing continues to other components in the tree. In this manner, reliability models are "bubbled" up to the highest level component which defines the model for the system.

Instantiation of RMAS modules proceeds in a top-down fashion with respect to the unreliable condition defined at the highest level. If aggregation were to proceed bottom-up without regard to the high level unreliable condition, the resulting model may contain transitions and state space elements that are irrelevant to the unreliable condition being analyzed.

Reliability models returned from each level in the problem-solving hierarchy adhere to the same format. That is, each model defines, for the component, the characteristics of its outputs given a relationship between the characteristics of its inputs and the states of the components (as a result of failures). Also, transitions define changes to the component as a result of failures or recoveries from failures (FDIR schemes).

The next section describes the RMAS modules in more detail. Since there are several intricate details of this algorithm that inhibit a comprehensive description at this time, an overview of the tasks involved in RMAS is given

first. This is followed by specific design details underlying each task. The explicit algorithm for the RMAS is listed in Appendix B.

4.3.1. RMAS IMPLEMENTATION

Figure 12 shows the major tasks of RMAS, and indicates each of these specific details by a "special" note number. These special notes will be referenced in the general process description given below and discussed in detail in sections 3.3.2.x, where x is the note number. Initially, the algorithm for aggregating all lowest level models into a reliability model for the parent is described. Then, in section 3.3.2.7, the RMAS is augmented to handle aggregation of intermediate level reliability models.

First, the aggregate model for the parent component is initialized to the local reliability model of the subcomponent that generated the parent's output. The first task, task 1, is then invoked for this subcomponent. The function to be performed by task 1 is to define a new reliability model that includes all transitions that contribute to the output characteristic.

The subcomponent's local reliability model is composed of output characteristic definitions (OCD) for each different component output characteristic and transitions for each component failure mode. For a given output characteristic, task 1 first decomposes the OCD into disjunctive normal form clauses (note 1, figure 12) so that each clause

is delineated by a logical OR, and within a clause, conditions are delineated by logical AND. Each condition within each clause is then analyzed to find transitions that contribute to the output characteristic. Task 2 is called within task 1 for each condition in the clause. It returns an output characteristic definition (OCD) for the condition and a list of transitions that contribute to the condition. The parent's OCD is changed to reflect these new transitions found (see 3.3.2.3). After all conditions for the clause have been analyzed, the next clause is analyzed in the same manner.

Task 2, given a condition to analyze, determines the condition type as either a component state (i.e., non-failed, failed via failure-mode-1, failure-mode-2, etc.), an input characteristic (i.e., GOOD, BAD, NIL), or an input predicate. We will defer the third condition type of "input predicates" to section 3.3.2.4. If the condition is a failure state, then a transition explicitly represents the component's transition to the failure state. This transition has the form:

IF <component>_NOF TRANTO <component>_<x> by <x>_RATE

where <component> is the name of the component, _NOF identifies a not-failed state, and <x> uniquely identifies the failure mode. This transition is returned from task 2.

If the condition is an input condition, then in order to find the transitions (and OCD) that contribute to the

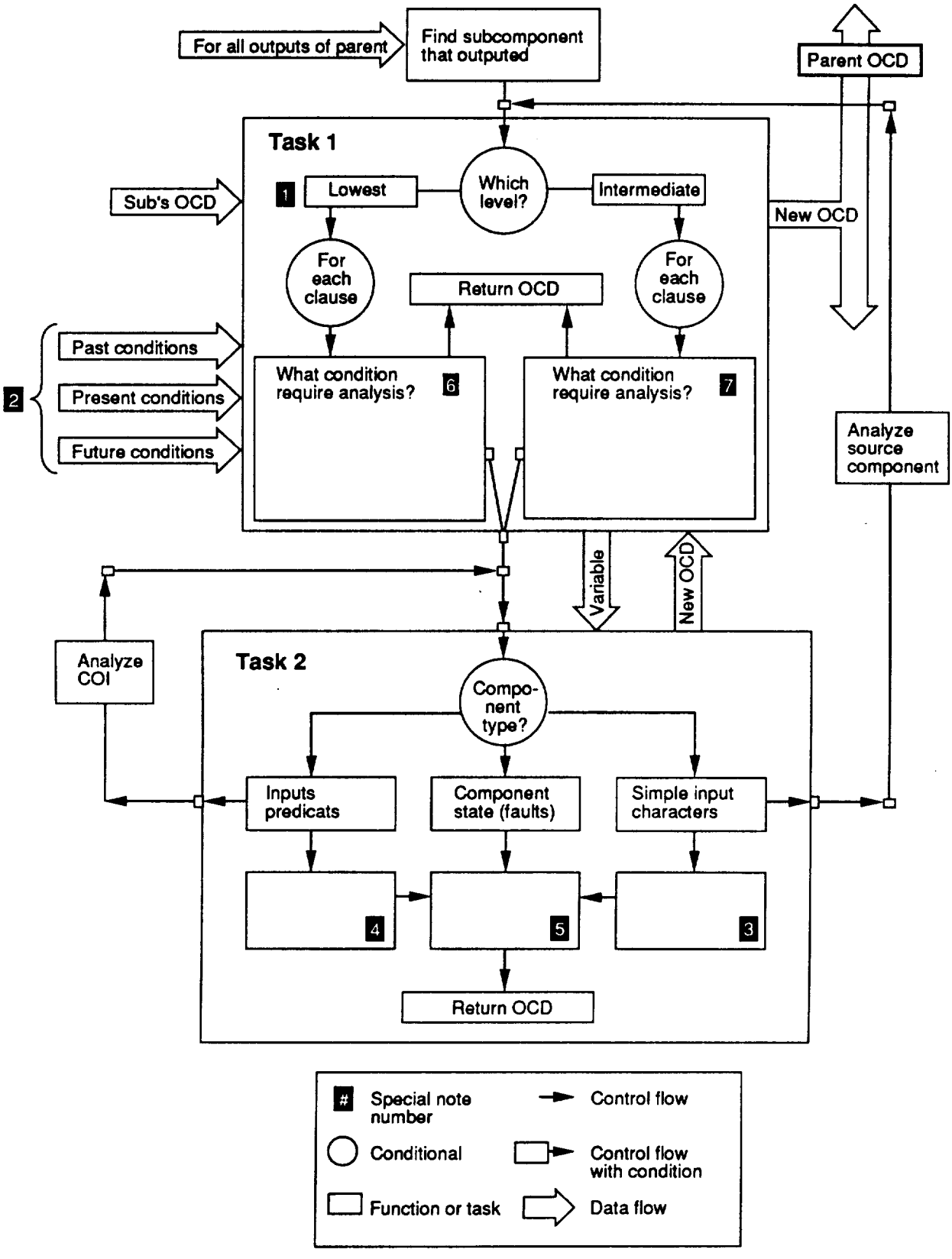


Figure 12. General Overview of RMAS

input condition, task 1 is called for the subcomponent that outputted the condition. Task 1 is, therefore, called recursively from within task 2, which was in turn called within task 1. At this time, it is useful to distinguish the invocations of task 1 (see figure 13). Let the subcomponent being analyzed by the first invocation of task 1 be referred to as the INPUTTING_COMPONENT since it inputs the data characteristic common to the subcomponents, and let the subcomponent of the latter invocation of task 1 be referred to as the OUTPUTTING_COMPONENT since it outputs the data characteristic to the INPUTTING_COMPONENT. Thus, task 2 of the INPUTTING_COMPONENT invokes task 1 for the OUTPUTTING_COMPONENT. Task 1, given the output characteristic for the OUTPUTTING_COMPONENT, performs as before. The new OCD, for the output characteristic, is decomposed into clauses, each condition of each clause is analyzed separately by task 2 to find transitions that contribute to the condition. As before, the conditions in each clause may represent component states for which simple transitions are defined or input conditions, for which task 2 will invoke a new task 1. This recursive process proceeds until there are no more input conditions to analyze, the input of the parent component is reached, or until a cycle is detected. Section 3.3.2.6 discusses the mechanisms for handling cycles.

The returned transitions from the OUTPUTTING_COMPONENT's task 1 to task 2 of the INPUTTING_COMPONENT are defined in terms of input conditions

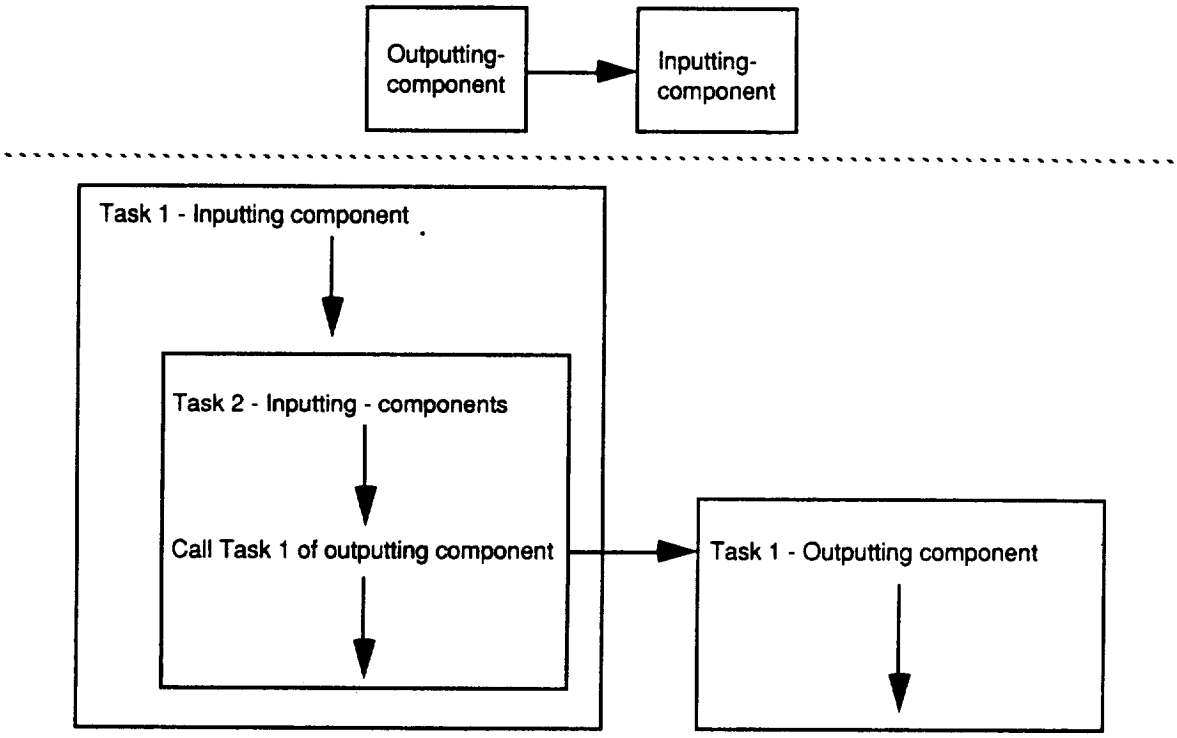


Figure 13. Distinguishing Invocations of Task 1

and state conditions for the OUTPUTTING_COMPONENT. Task 2 may change the transitions returned to reflect additional changes to the INPUTTING_COMPONENT. The decisions as to what changes are made to the transitions and when such changes are made is discussed in section 3.3.2.3. RMAS attempts to change the model so that the least number of variables is used by substituting for some output characteristics, an equivalent expression which defines that characteristic in terms of other variables. By choosing carefully which variables will be referenced in the final model, and which will be inferred by the other variables (see 3.3.2.3), RMAS reduces the number of variables which will eventually be state space vector elements. This model reduction is not optimum in terms of the number of states generated by a model. Further techniques are applied to the resulting global model of the system in the Model Reducer/Encoder.

Task 2 returns the transitions to the instance of task 1 from which it was invoked. Task 1 proceeds with another condition, another clause, or returns to the task 2 from which it was invoked. Eventually, the initial top level task 1 is reached and the aggregate model for the parent component is defined.

4.3.2. SPECIAL NOTES

The preceding discussion outlined the general flow of tasks for RMAS. Implementational details omitted there are

discussed in the next sections. Each of the following sections (3.3.2.1 through 3.3.2.7) corresponds to a special note on figure 12. The following discussion may also serve as an introduction to the documentation of the task algorithms for RMAS, contained in appendix B, or as a guide to the trace of the example reliability analysis problems given in appendices C and D.

4.3.2.1. DISJUNCTIVE NORMAL FORM

In Task 1, an OCD is first reordered into disjunctive normal form (DNF). Enforcing this ordering has several advantages that will be introduced here and elaborated in subsequent sections. First, when transitions are found that contribute to a condition in the OCD, the conditions in the OCD are sometimes changed in task 2. By ordering the OCD in DNF form, the changes made to a clause are localized to that clause (The algorithms for changing the OCD require that the OCD be in DNF form). Second, as will be seen in section 3.3.2.6. DNF clauses allow detection of cycles in the analysis. Third, DNF provides a uniform ordering that minimizes ambiguity.

To translate a boolean expression into DNF form, the following rules are applied iteratively on the expressions.

1. A AND B --> one clause

   | A AND B

2. X OR Y --> separate subclauses

| X

| Y

3. (X OR Y) AND Z -->

| X AND Z

| X AND Z

where | delineates the clause boundaries, A and B are individual conditions, and X and Y are subclauses or individual conditions.

In the absence of parentheses, it is assumed that that logical OR has the lowest precedence, logical AND is next, and boolean comparators (<, >, =) have the highest precedence. Therefore,

A OR B > C AND D is equivalent to:

A OR ((B > C) AND D) which, in DNF is:

| A

| (B > C) AND D

4.3.2.2. ORDER CONDITIONS FOR ANALYSIS

Task 1 must analyze all conditions in all clauses of an OCD in order to find the OCD and all transitions that contribute to an output characteristic. To maintain the status of analysis of each condition, consider 3 global queues. (The queues are global for simplicity of description. In fact, the queues may be partitioned into subsets for each

subcomponent). To keep track of the order of analysis, task 1 maintains a queue of conditions to be analyzed - called the FUTURE_CONDITIONS queue. Task 1 removes one condition at a time from this queue and places the condition on the PRESENT_CONDITIONS queue. Once a condition has been analyzed, it is simply removed from the PRESENT_CONDITIONS queue and placed on the PAST_CONDITIONS queue and another condition is taken from the FUTURE_CONDITIONS queue. If the same condition appears in two or more clauses, then this condition need not be analyzed in the second clause. In other words, the condition is on the PAST_CONDITION queue, and it has already been determined what transitions contribute to the condition. Therefore, the condition is removed from the PRESENT_CONDITIONS queue (it remains on the PAST_CONDITIONS queue).

If the subcomponents within a parent interact in a cycle, then, in analyzing the input conditions for a subcomponent, there will exist two invocations for that subcomponent on the stack of recursive invocations of Task 1. It may be that the second invocation of Task 1 for the subcomponent is the same condition currently under analysis in the first invocation (thus the condition is on the PRESENT_CONDITIONS queue). Since the goal of the analysis is to find transitions that contribute to a condition, we can conclude that there are no transitions that contribute to the condition through the current path. Therefore, task 1 for the second invocation returns no transitions. We

defer further discussion of this situation to section 3.3.2.6.

In summary, the three queues are used to establish an orderly analysis of the conditions within clauses for an OCD, to prevent repeated analysis of conditions that had been analyzed in previous invocations of Task 1, and to detect a cycle.

4.3.2.3. COMBINE OUTPUT CHARACTERISTIC DEFINITIONS

As discussed in the general overview, task 2 invokes task 1 for another subcomponent (OUTPUTTING_COMPONENT) in the analysis of an input condition for the subcomponent under its domain (INPUTTING_COMPONENT). Task 1 for OUTPUTTING_COMPONENT returns an OCD, which includes a set of contributory transitions, for that input condition of the INPUTTING_COMPONENT. The transitions returned from the OUTPUTTING_COMPONENT's task 1 to task 2 of the INPUTTING_COMPONENT are defined in terms of input conditions and component states for the OUTPUTTING_COMPONENT. Task 2 must aggregate the OCD and transitions returned from the OUTPUTTING_COMPONENT with the OCD of the INPUTTING_COMPONENT so that changes to all affected conditions are represented in the transitions. To illustrate this, consider the following (illustrated in figure 14):

INPUTTING_COMPONENT (A):

OUTPUT y:b IF x:b and q:b and A_FAIL

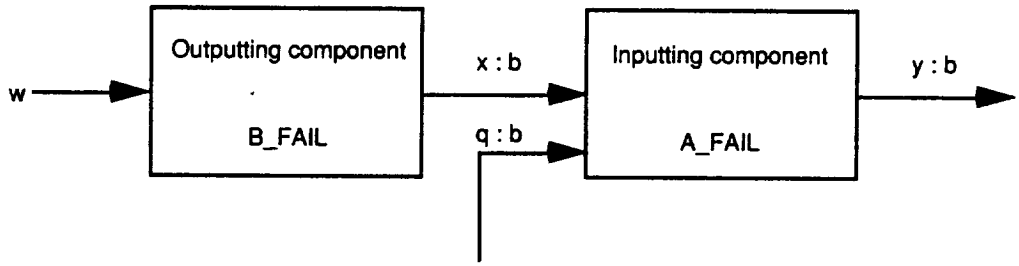


Figure 14. Inputting vs Outputting Components

< transitions >

OUTPUTTING_COMPONENT (B):

OUTPUT x:b IF w:b and B_FAIL

IF B_NOF TRANTO B_FAIL

The INPUTTING_COMPONENT, A, is analyzing its OCD for y:b and has called task 1 for OUTPUTTING_COMPONENT, B, to find any transitions that contribute to A's input condition x:b. The OCD returned from B has a single transition that represents a failure of component B, and the OCD for x:b represents how B's failure affects its output x. (Of course, the input w:b would also have to be analyzed, but assume for this illustration that no transitions were found to contribute to w:b). Task 2 of component A must aggregate this information to define the affect of the transition found (i.e., B's failure) on A's output characteristic y:b. There are two options:

1. Replace the reference to x:b in A's output characteristic definition with its equivalent definition in B's output characteristic definition. Do not change the transition returned from B.

AGGREGATE OCD for parent of A and B:

OUTPUT y:b IF (w:b and B_FAIL)

and q:b and A_FAIL

IF B_NOF TRANTO B_FAIL

2. Change the transition returned from B to reflect a change to x:b also. Do not change A's OCD.

AGGREGATE OCD for parent of A and B:

OUTPUT y:b IF x:b and q:b and A_FAIL

IF B_NOF TRANTO B_FAIL

IF w:b TRANTO x:b

ENDIF;

When option 1 is used, the variable replaced is put on a list (called LOGICAL list) with its equivalent representation that replaced it in the OCD. If the variable is referenced again in another clause (or in another invocation of task 1 for the subcomponent), the logical equivalent is substituted in for the variable. A corresponding list called BASES is used to keep track of those variables that are referenced directly in the OCD or transitions for the aggregate model.

To reiterate, the list BASES will be used to identify those conditions that will be referenced in the OCD and in the transitions, and the list LOGICAL will identify the conditions that will not be referenced in the OCD or transitions. LOGICAL list will also identify the combination of BASES that are equivalent to the logical condition. All variables will be on the BASES list except if they are substituted by a logical equivalent of other variables using option 1. Option 1 is advantageous in that it eliminates

reference to an intermediate variable in the aggregate model. This reduces the state space size of the aggregate model and eventually for the overall model of the system. It also involves a simple substitution, whereas the algorithm for option 2 is not as straight forward (see procedure MODEL_COMBINATION_2 in the algorithm for RMAS.). Therefore, whenever possible, task 2 invokes option 1. However, some situations require the use of option 2. These situations occur in the specification of non-fault transitions (to be discussed in 3.3.2.6) and in the specification of input predicates (to be discussed in note 3.3.2.4). In both these situations, an internal variable is found which must be represented explicitly, and therefore option 2 is used to combine models between subcomponents instead of option 1. Figure 15 shows a revision of RMAS in which task 2 is augmented with options 1 and 2.

4.3.2.4. INPUT PREDICATES:

For some components, defining an OCD in terms of simple boolean relationships among input characteristics is not sufficient. This is especially true when inputs are redundant and the effects of single and multiple redundant component failure modes on other components are analyzed. It is therefore necessary to allow input characteristics to be defined by predicates. For example, define a VOTER as a component that outputs the majority value of the inputs it receives from redundant components. In specifying the OCD for the voter's corrupted output as a function of its

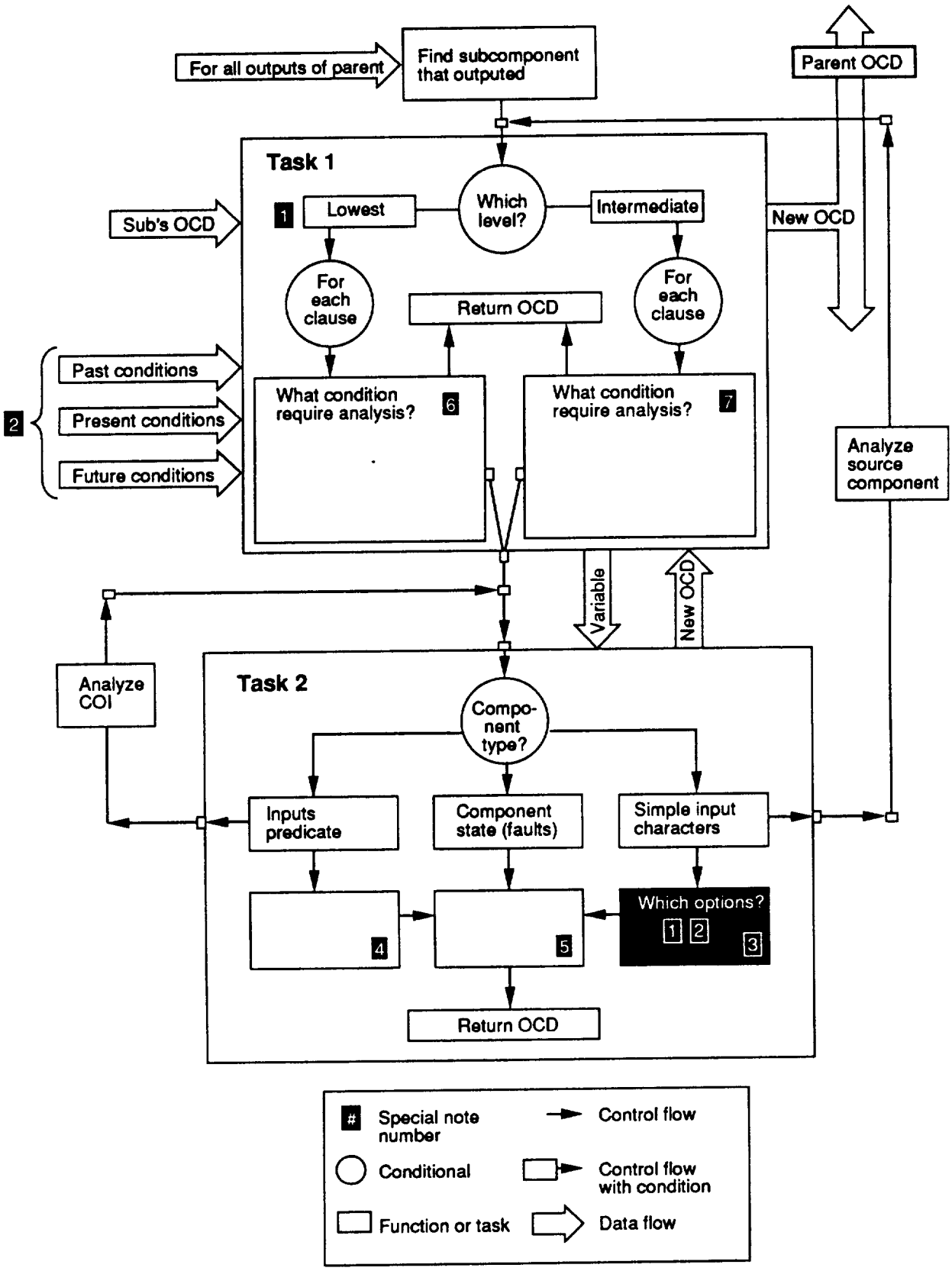


Figure 15. Model combination

corrupted inputs, one might define the "number" of inputs that are corrupted (i.e., #x(i):b for all x(i) inputs to the voter). By allowing this specification, the output definitions for components may contain conditions of input predicates. In the case of a voter, the OCD may be:

```
OUTPUT y:b if #x(i):b > #x(i):g
```

This states that the voter's output, y, is corrupted if the number of corrupted inputs exceeds the number of non-corrupted inputs. In the absence of the # predicate, an equivalent specification must be defined as:

```
OUTPUT y:b IF x1:b and x2:b
          OR x1:b and x3:b
          OR x2:b and x3:b
          OR x1:b and x2:b and x3:b
```

for a voter component with three inputs. Thus, being able to specify input predicates is beneficial. Further, the final aggregated model benefits from specifying the state space elements in terms of predicates. ASSIST contains a rich set of primitives for specifying state space elements and transitions that change those elements. For example, the "#" predicate may be represented as a single state space integer whose range is from zero to three. A possible ASSIST syntax for this is:

```
X:[0..3]   {state space}
```

```
DEATH IF X < 2.  {specification of unreliable states}
```

where x represents the number of inputs that are not corrupted. The Death statement represents the state of the system in which the output of the voter is corrupted. Contrast this with the ASSIST model:

```
STATE = X1[0..1], X2[0..1], X3[0..1]

DEATH IF (X1=1 AND X2=1)
     OR (X1=1 AND X3=1)
     OR (X2=1 AND X3=1)
     OR (X1=1 AND X2=1AND X3=1)
```

where X1, X2, and X3 represent whether or not inputs 1, 2 or 3 respectively is corrupted.

In order to specify input predicates, RMAS requires guidance in finding transitions that contribute to predicates. Therefore, internal to RMAS are a set of templates, each template indicating how a predicate should be analyzed and represented in the aggregate model. For example, the model of the # predicate might consist of:

```
MODEL:  #(<cond>)

     COI: <cond>

     Change rep:  no

          Contributory transitions:

               replace IF (<cond>)
```

```
                    with IF #(<cond>) > 0
               replace TRANTO <cond>
                    with TRANTO #(<cond>)=#(<cond>) + 1

          Detractory transitions:
               replace <cond>
                    with #(<cond>)=#(<cond>) - 1

     ASSIST CONVERSION:  #(<cond>) : integer
```

The template first specifies the condition of interest (COI). The COI is a simple input characteristic that is to be traced to find transitions that contribute to the predicate. The COI for #(X(i):b) is X(i):b - find transitions that contribute to a particular input to the voter being corrupted. The contributory transitions section indicates the changes that are to be made to the transitions found for the COI to reflect changes to the predicate. Here, the transition:

```
IF X(i):B TRANTO ...
```

is changed to:

```
IF #(X(i):b) > 0 TRANTO ...
```

Similarly, the transition:

```
If ... TRANTO x(i):b
```

is changed to:

```
IF ... TRANTO #(X(i):b) = #(X(i):b) + 1;
```

A similar change is defined for detractory transitions. The definition and use of detractory transitions is described in the next section. Figure 16 shows the change to task 2 to support input predicates.

The template also defines the ASSIST representation of the function to be used in Process 4 to encode the aggregate system model into ASSIST primitives. For the "#" predicate, the template defines a state space element whose name is indicative of the condition and the function and whose type is integer. Further specification may be added to define the bounds on the integer.

Finally, the change rep{resentation} field indicates that a predicate can not explicitly be represented in ASSIST syntax, and therefore, a representation in ASSIST using a different predicate must be substituted. ASSIST allows integer state vector variables, and therefore, the predicate # can directly be represented in ASSIST using an integer variable as shown above. However, the predicate, ALL, for example, can not be represented in ASSIST directly. Therefore, it would be encoded using an integer variable also by comparing the number to the total number of elements possible (e.g., ALL(X(i):b) becomes #(X(i):b) = N where N is the number of X(i)). The template for ALL therefore includes a "change rep" field that refers to the "#" predicate:

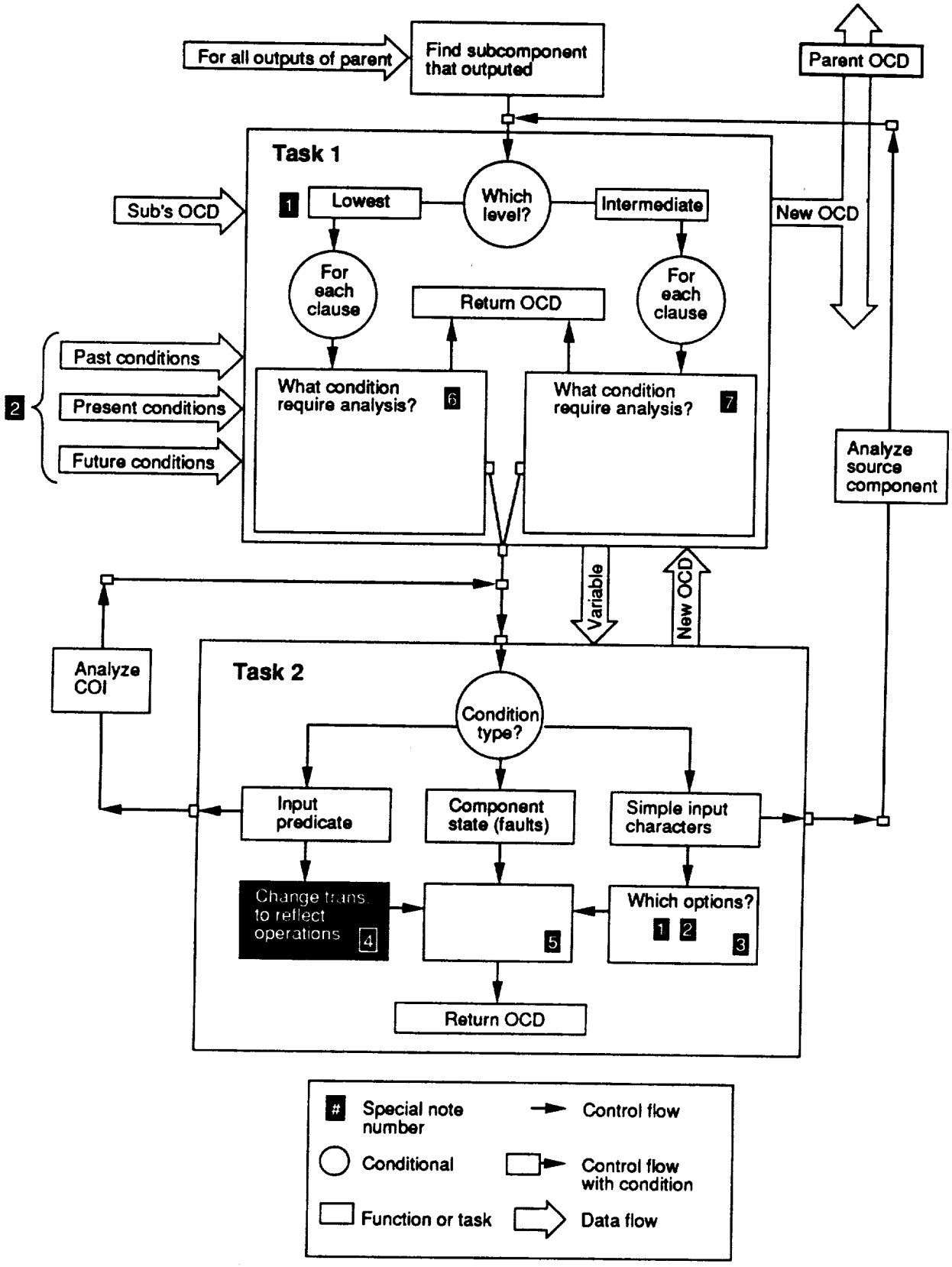


Figure 16. Input Predicates

MODEL: ALL(<cond>)

COI: <cond>

Change rep: #(<cond>)=?<query user> = max(obj) from SYSD where obj is component (or component of data) in <cond>

Contributory transitions:

replace IF (<cond>)

with IF #(<cond>) = Max(obj)

replace TRANTO <cond>

with TRANTO #(<cond>)=#(<cond>) + 1

Detractory transitions:

replace <cond>

with #(<cond>)=#(<cond>) - 1

ASSIST CONVERSION: #(<cond>) : [0..max(obj)]

Recall from section 3.3.2.3 that transitions and the OCD returned from analysis of an input condition are aggregated with the current OCD from the inputting component. Two options were described for this aggregation, with the preferred option (option 1) replacing the input characteristic in the aggregated model with an equivalent definition from the OCD of the outputting component. However, if the input characteristic is a COI for a predicate, it must remain in the model so that the transition may be changed to reflect the predicate. In this situation, option 2 must be

used to combine models in order to preserve the COI variable until the reference to the the variable is replaced by the changes as specified by the contributory and detractory transitions fields of the input predicate.

4.3.2.5. DETRACTORY TRANSITIONS:

Task 1 iteratively calls task 2 to find transitions that contribute to a condition in each clause of the OCD. A given transitions may also detract from the presence of other conditions, and this effect must be reflected in the transition also. For simple conditions such as simple input characteristics that are modeled as bases and for component states, no changes are necessary to reflect detractory transitions.

For example:

```
IF P_NOF TRANTO P_BAD ...
```

is a contributory transition for PBAD, but it is also a detractory transition for PNOF. Since it is implicitly assumed that a component can not be in more than one state simultaneously, there is no need to change the transition to:

```
IF P_NOF TRANTO P_BAD, NOT(P_NOF) BY P_BADRT
```

The same holds true for simple input conditions that are modeled as bases:

IF x:(g) TRANTO x:(b) ...

But for predicates (#(<cond>), for example), the detraction is subtle:

For Example:

IF PNOF TRANTO PBAD, #(x:(b)) = #(x:(b)) + 1 BY ...

Recall that #(x:(b)) and #(x:(g)) are base conditions. When #(x:(b)) increases by 1, #(x:(g)) decreases by 1. To reflect this change:

IF PNOF TRANTO PBAD, #(x:(b)) = #(x:(b)) + 1
#(x:(g)) = #(x:(g)) - 1 BY ...

To accomplish this, a procedure, FIND_DETRACTORY_TRANSITION, is called in task 2 for each condition. The procedure determines if any previously defined transitions detract from the current condition (if the current condition is a predicate) and if any currently defined transitions detract from previously defined predicate conditions. Figure 17 illustrates this addition to task 2.

4.3.2.6. NON-FAULT TRANSITIONS:

As discussed in the general overview, in order to find transitions that contribute to an input condition, the components are traced in reverse order of the flow of information between components so that an input condition is traced to the component responsible for that condition. That

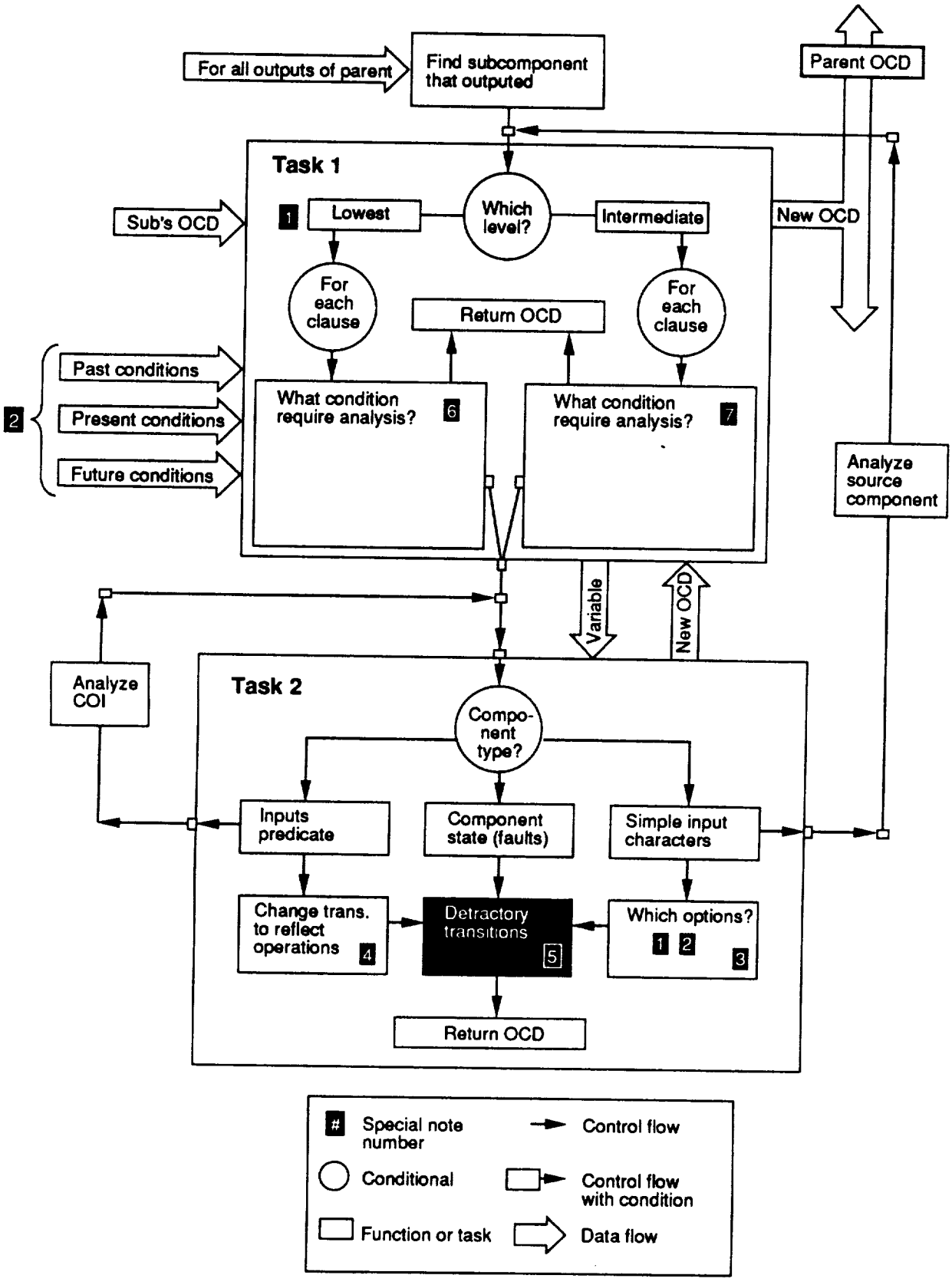


Figure 17. Detractory Transitions

component's output condition may depend on its input conditions, and so forth until the input of the parent component is reached or a cycle is detected. This section discusses the implications of cycles in the analysis and procedures for handling them.

Recall that when task 1 is called to analyze an output characteristic, the output characteristic definition (OCD) is first translated into disjunct normal form so that each clause is delineated by a logical OR, and within a clause, conditions are delineated by logical AND. We showed that all OCD's could be translated into an equivalent DNF form. Once in DNF form, task 1 substitutes in for variables on the LOGICAL list, their base equivalent expression. As a result, there are five possible DNF clause categories:

OUTPUT R	R	case 1
	R AND A	case 2
	^R AND A	case 3
	^R	case 4
	A	case 5

where R is the output condition, ^R represents NOT(R) and A represents the rest of the clause which does not include a reference to R.

Cases 1-4 occur as a result of the substitutions of BASES for LOGICAL conditions and indicates that a cycle in the analysis has occurred. Case 5 is the normal OCD that has been discussed thus far in which the output effect R occurs

in the presence of other conditions (input conditions and component faults).

4.3.2.6.1. CASES 1 AND 2:

In cases 1 and 2, the effect R occurs on both sides of the conditional, indicating that within the cycle, no change to R has occurred. Since the purpose of the analysis is to find transitions that contribute to the occurrence of condition R, case 1 and 2 clauses indicates that there are no transitions that contribute to R as a result of this sequence of conditions among the subcomponents in the cycle.

Case 2 differs by case 1 only in the presence of other conditions A in the output characteristic definition for R. Analysis of the conditions within A need not be performed since the OCD indicates that the condition R is true a priori (as evidenced by the inclusion of R in the OCD. Therefore, analysis of conditions within A would not be beneficial in the pursuit of transitions that contribute to the occurrence of R for this clause. Thus, the output clause "R AND A" may be ignored, and the conditions within A not analyzed. In order to ensure the correctness of transitions found in other clauses for the OCD of R, any transitions returned by a case 5 type clause should be changed in task 1 to reflect that the R condition must not be true for the transition to occur:

Change IF A TRANTO ...

To IF A AND ^R TRANTO ...

for all transitions found in other clauses.

This change appears similar to the changes that are made in task 2 to aggregate two component models (option 2). To differentiate the use of these two routines, figure 19 illustrates an example in which both option 2 of task 2 and case 2 of task 1 change the aggregate transition. Task 1 of the INPUTTING_SUB's OCD analyzes two clauses of the its output characteristic R. The first clause adheres to case 2 which indicates that a cycle is detected from which no transitions have occurred, and is not analyzed. The second clause, however, is still analyzed. Let the condition B in clause 2 be an input condition. Task 2 calls task 1 for the OUTPUTTING_SUB of B and it returns an OCD for B. Task 2 of the inputting component then uses option 2 to collapse the OCD returned into an equivalent transition. Task 2 returns this transition to task 1 which removes the first clause from the output characteristic definition and adds a ^R condition to the rest of the transitions.

4.3.2.6.2. CASES 3 AND 4:

In cases 3 and 4, the effect ^R occurs on the conditional side of the OCD indicating that somewhere in the cycle, a transition changed the condition from ^R to R. Since the analysis at each subcomponent in the cycle identifies explicitly any transitions that result from component failures (i.e., changes in component state), this transition

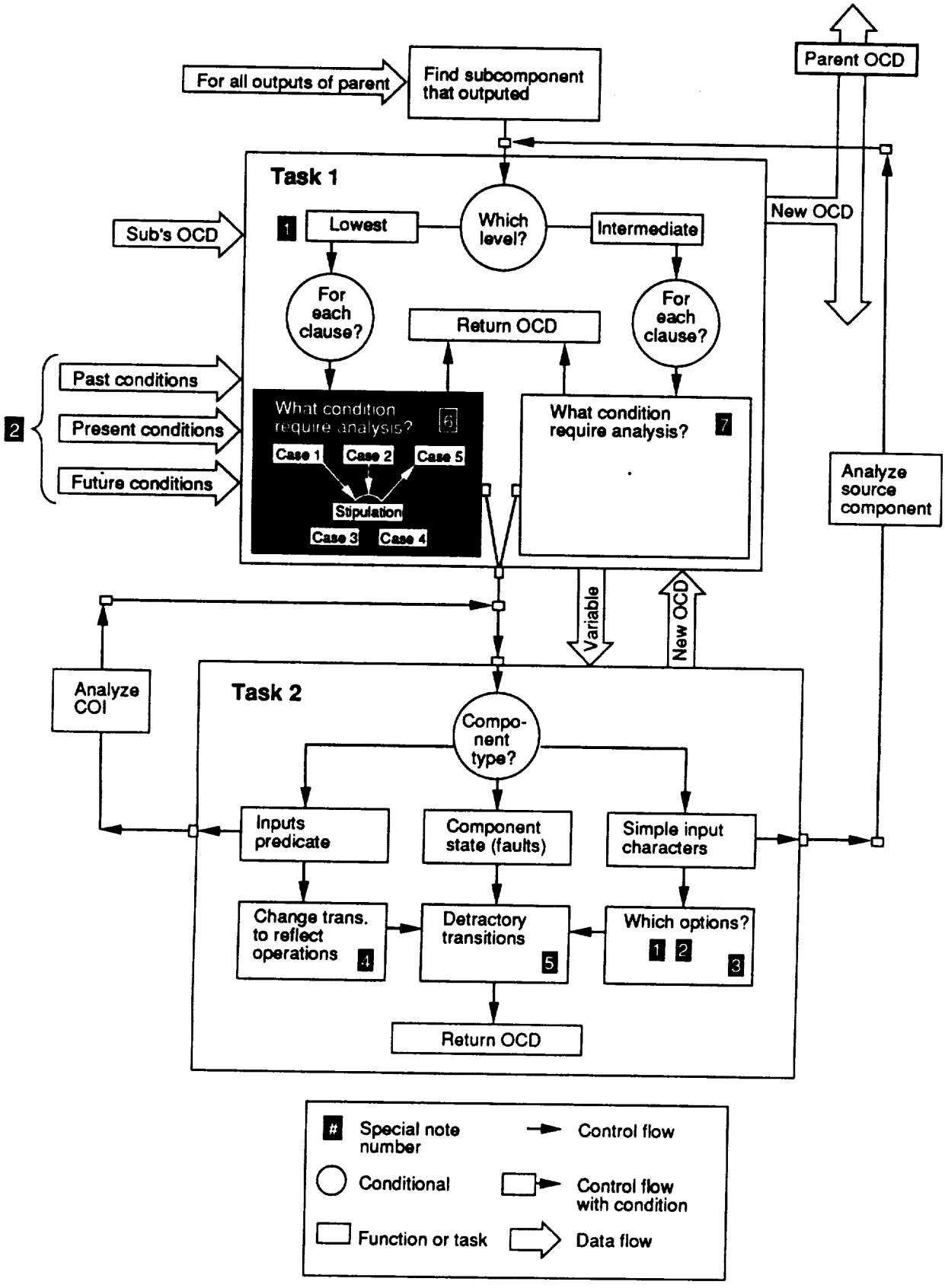


Figure 18. Non-Fault Transitions

must be a non-fault related transition. An example of a non-fault transition is a recovery transition, (i.e., some change in the operational state of the components as a result of failures). A non-fault transition is handled in the following way:

1. Analyze the rest of the clause (A in case 3). Any transitions returned here are not subject to the changes mandated by other clauses (cases 1 and 2).

2. Add a new transition to reflect the change of state from ^R to R

IF A and ^R TRANTO R BY T(CYCLE)

Because the clauses are ordered in DNF form, the least number of conditions necessary to define the non-fault transition is localized in one clause. The rate T(cycle) will be represented in ASSIST as a mean and standard deviation of the cycle time. This value must be entered by the user before the SURE model is generated, just as the failure rates of the components must be entered by the user.

If a non-fault transition is returned from an OUTPUTTING_COMPONENT to an INPUTTING_COMPONENT, option 2 will be used to aggregate the models so that the loop variable (R) is not eliminated from the state space. Furthermore, option 2 need not be applied to the non-fault transition since the transition already references R.

4.3.2.6.3. CASE 5:

Case 5 clauses are ones in which the OCD does not reference the output effect R or ^R as a condition in the clause. This is the typical clause that are present in the absence of cycles. Each condition in this clause is analyzed separately (through the invocation of task 2) and results are combined as specified in section 3.3.2.3. Transitions returned from the analysis of these conditions may be changed to reflect the elimination of any type 1 or type 2 clauses for the same output characteristic. This was discussed in section 3.3.2.6.1.

4.3.2.6.4. CASE SUMMARY:

The following summarizes the clause analysis just described.

CASE 1: OUTPUT R IF R

1. remove clause from output characteristic definition

2. in order for transitions from other clauses to be converted to affecting R, they must include:

   IF ... and ^R ... TRANTO

CASE 2: OUTPUT R IF R AND A

1. Do not analyze A.

2. Remove clause from final output characteristic definition for R.

3. in order for transitions from other clauses to be converted to affecting R, they must include:

IF ... and ^R ... TRANTO

CASE 3: OUTPUT R IF ^R AND A

1. Add a non-fault transition:

IF A and ^R TRANTO R BY T(cycle)

2. Analyze A: The transitions returned are not subject to stipulation 3 of cases 1 and 2 since the non-fault transition already represents the transition to R.

CASE 4: OUTPUT ^R

1. Add a non-fault transition:

IF A and ^R TRANTO R BY T(cycle)

CASE 5: OUTPUT R IF <normal conditions that excludes R>

1. Analyze all conditions [1]

2. Any transitions returned are subject to the changes specified by cases 1 and 2 of other clauses.

Figure 19 illustrates the changes to task 1 to address non-fault transitions. A detailed example of the use of these cases is given in appendix D (example 2).

---

1 Recall from section 3.3.2.2 that only those conditions not on PAST_CONDITIONS queue are analyzed.

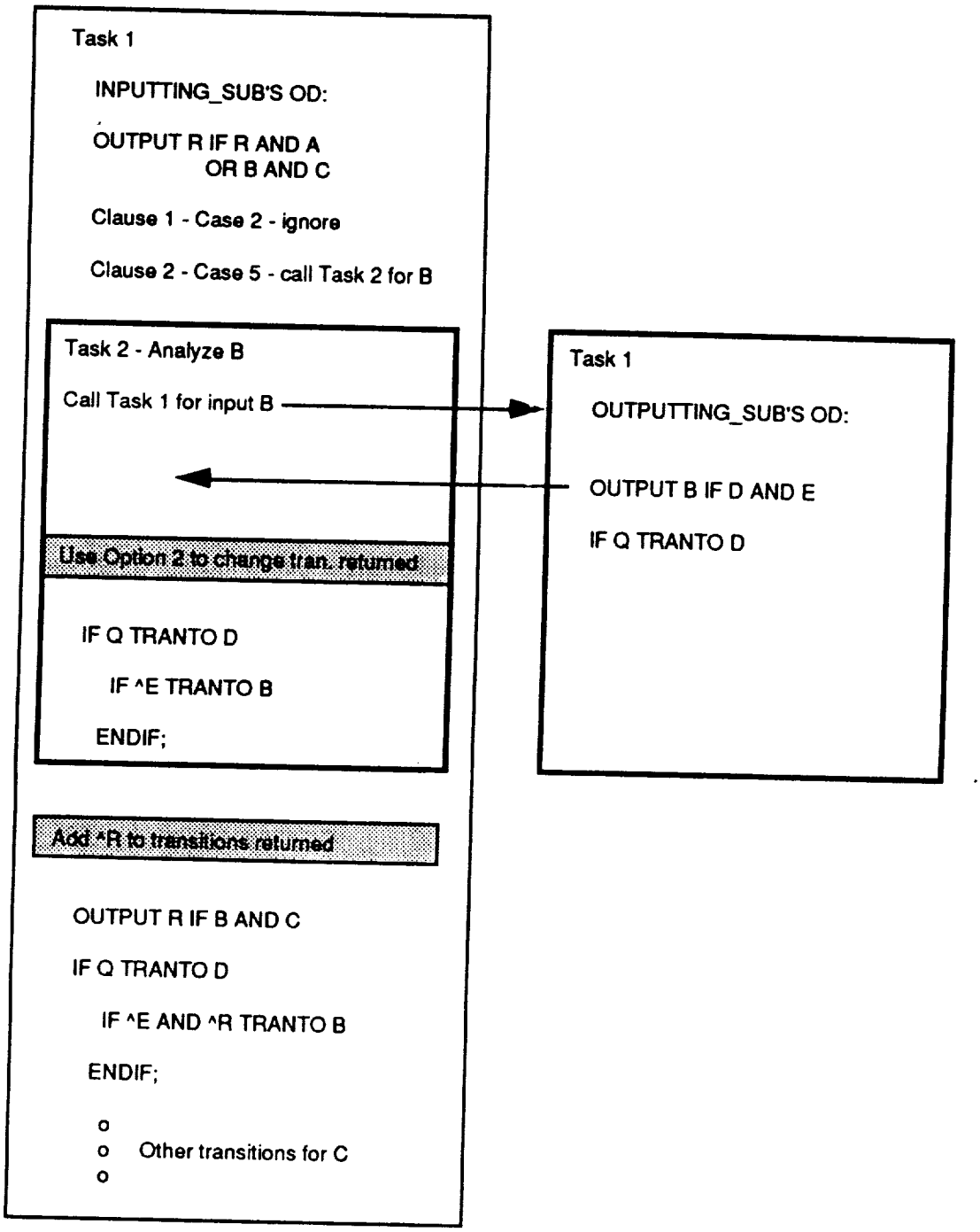


Figure 19. Illustration of Changes to Transitions

4.3.2.6.5. MODELING RECOVERY TRANSITIONS - ANALYTICAL BASIS

The detection of cycles that involve "unspecified" transitions is the only mechanism for specifying fault recovery transitions. It is assumed, therefore, that any fault recovery would involve more than one component and that those components interact in a cycle. This is a reasonable assumption since a typical FDIR scenario involves a component or components that fail, some mechanism that detects the failure and depending on the FDIR scheme, takes some action to modify the effects of the failure at some place in the system. Through detection of cycles, RMAS identifies that faulty effect whose behavior is modified by the recovery action. Thus FDIR schemes are cyclic by nature. Even self-repairing components have a substructure consisting of a separate detection/recovery component whether that "component" is software or hardware. One alternative method of modeling fault recoveries uses a coverage number - representing the probability that a system in a failed state transitions to non-failed state. In this case, the recovery could be modeled similar to a component failure mode:

IF P_BAD TRANTO P_RECOVER BY <rate>

This is sufficient for many models in which the recovery mechanism itself is not modeled.

4.3.2.7. INTERMEDIATE LEVEL MODELS:

This final section concerns the invocation of RMAS to combine component models that are themselves combinations of lower level models. A subcomponent model to be combined is an aggregation of lower level subcomponents also.

For example, in figure 20, to define an aggregate model for component P, the subcomponent models for components X, Y and Z are combined. Components X, Y, and Z have models that are themselves aggregates of subcomponent models from earlier invocations of RMAS. Component X, for instance, has a model that is an aggregate of subcomponents A and B. This aggregate model was defined by RMAS for component X.

Each model for X, Y, and Z has an OCD for each of its outputs and transitions that define failure events and failure recoveries between subcomponents within X, Y, and Z. RMAS, for component P, must analyze these models and define and OCD for each of P's outputs and transitions that reflect changes to P's state.

The models (OCDs and transitions) for X, Y, and Z contain three types of conditions:

1. subcomponent state conditions (failures for A, B, C, etc.),

2. input and output conditions that are also visible to the parent component, P (e.g. input condition q, s, u, and w in figure 20).

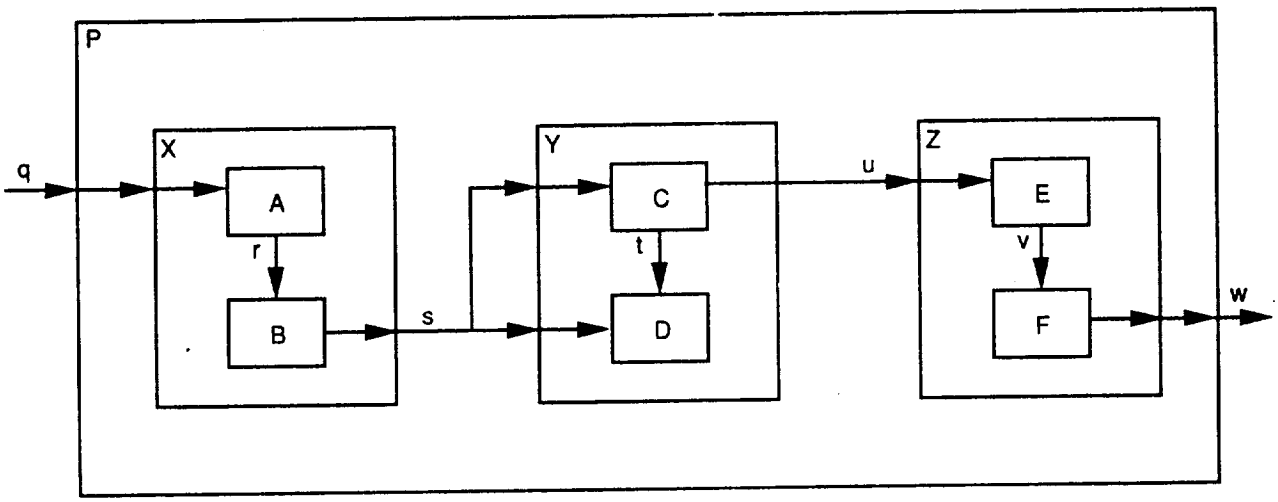


Figure 20. Intermediate Level Component analysis

3. input conditions between A and B, C and D, or E and F, that are not visible to the parent component, P (e.g., r, t, and v).

As before, the key to defining an aggregate model is to trace the interrelationship between the models (e.g. via conditions q,s,u, and w). However, these conditions are not always present in the OCD alone. Rather, it is possible that some are only referenced in the transitions for other conditions. Therefore, the transitions that have been defined for the lower level models must be searched, so that all conditions contributing to the output characteristic are traced.

The OCDs and transitions returned from analysis of conditions in transitions are subject to option 1 or option 2 rules, except that the substitutions will be made on the transitions instead of on the parent OCD.

This change is implemented as follows: RMAS maintains for each condition, a list of transitions that contribute to and detract from the condition (the contributory transition list (CTL) and detractory transition list (DTL)). For each OCD of the parent component, input conditions that are inputs to the parent component (e.g. u for component Z, s for component Y, and q for component X) are analyzed through task 2 as before. For state conditions in each clause (e.g. failure states of A, B, C, D, E, or F) or for any input conditions not visible to the parent component, P (e.g. r, t,

or v), the CTL and DTL lists are inspected and all transitions that contribute to and detract from these internal conditions are searched. The conditions part of such transitions may include parent input characteristics, internal (subcomponent) input characteristics, or subcomponent failure mode states.

Any parent input characteristic not yet analyzed causes an invocation of task 2 for their analysis.

For internal input characteristics or subcomponent failure mode states, the corresponding CTL/DTL lists are inspected and the search continues recursively for all parent input characteristics that need to be analyzed.

Figure 21 shows the RMAS algorithm with all extensions.

4.4. LOCAL MODEL GENERATOR

The Local Model Generator traces the effects of lowest level component failure modes on other components in the system by following the functional description of the components (defined in the BBD/SYSD). For each of the lowest level components in the SYSD, the Local Model Generator defines a local reliability model.

Stated formally: define the problem instance for the Local Model Generator (LMG):

Given

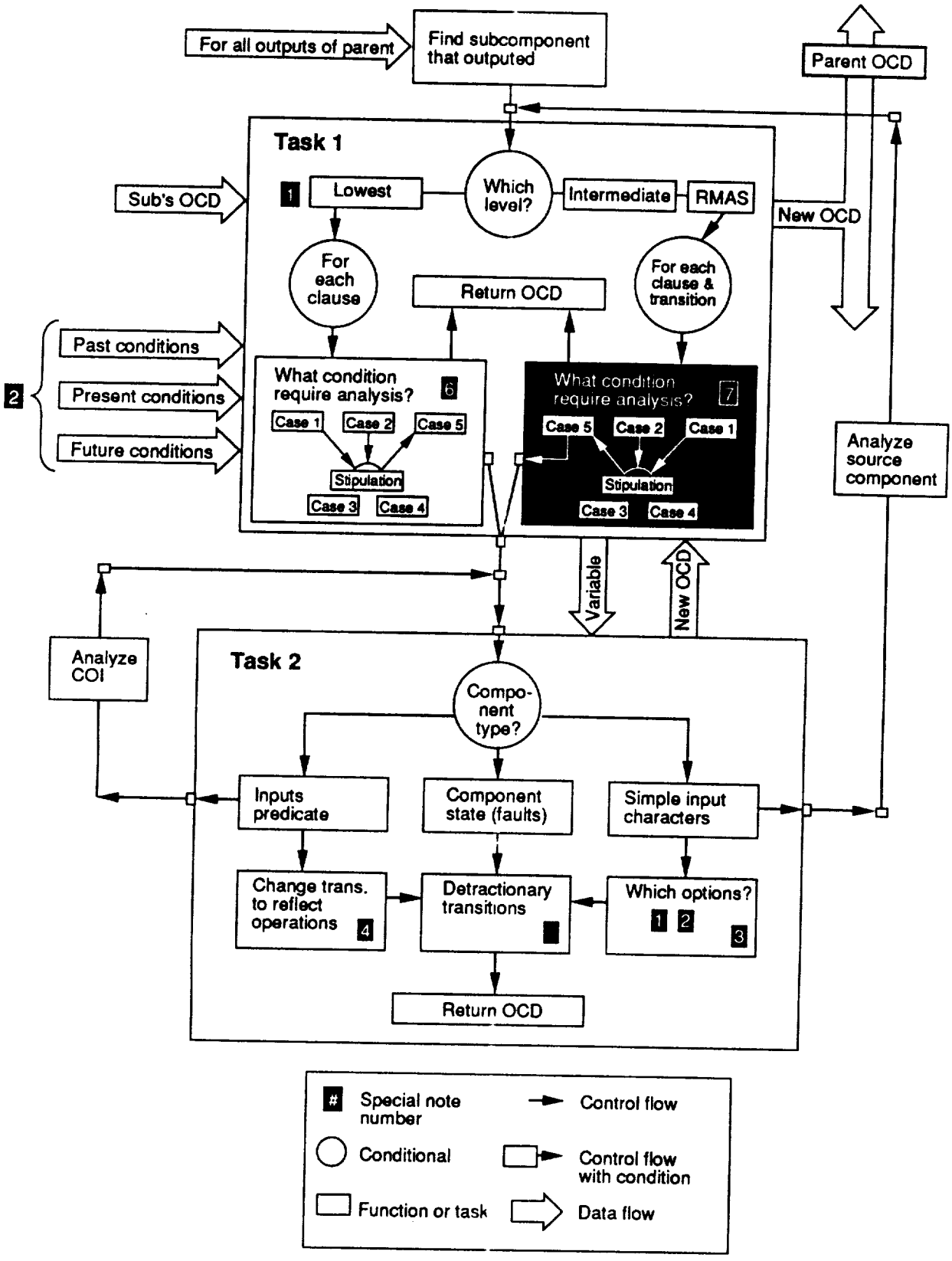


Figure 21. Intermediate Level Analysis

1. a hierarchical description of the candidate architecture (represented in the BBD and SYSD) in which component functions are defined down to the lowest level of interest.

2. a high level system unreliable condition for which the reliability is to be established,

define for each of the lowest level components, a local reliability model. Each local reliability model defines for the component all output characteristics as a function of the states of the component (i.e., failure modes) and the characteristics of the input to that component (e.g., corrupted or non-corrupted inputs).

Before detailing the process flow of the LMG in the next sections, it may be beneficial for the reader to review the function and failure mode specification of the intermediate and lowest level components defined in section 3.1.2.2 and the distinction between the RMAS modules and the LMG modules outlined in section 3.1.2.4.

4.4.1. PROCESS OVERVIEW:

The Local Model Generator consists of 3 processes: one process (I1) is invoked for the highest level and one for each intermediate level component, and two processes (L1 and L2) are invoked for each lowest level component. These are shown in figure 22. Like the RMAS processes, the LMG processes are instantiated in a hierarchy corresponding to the components defined in the SYSD (and supporting BBD).

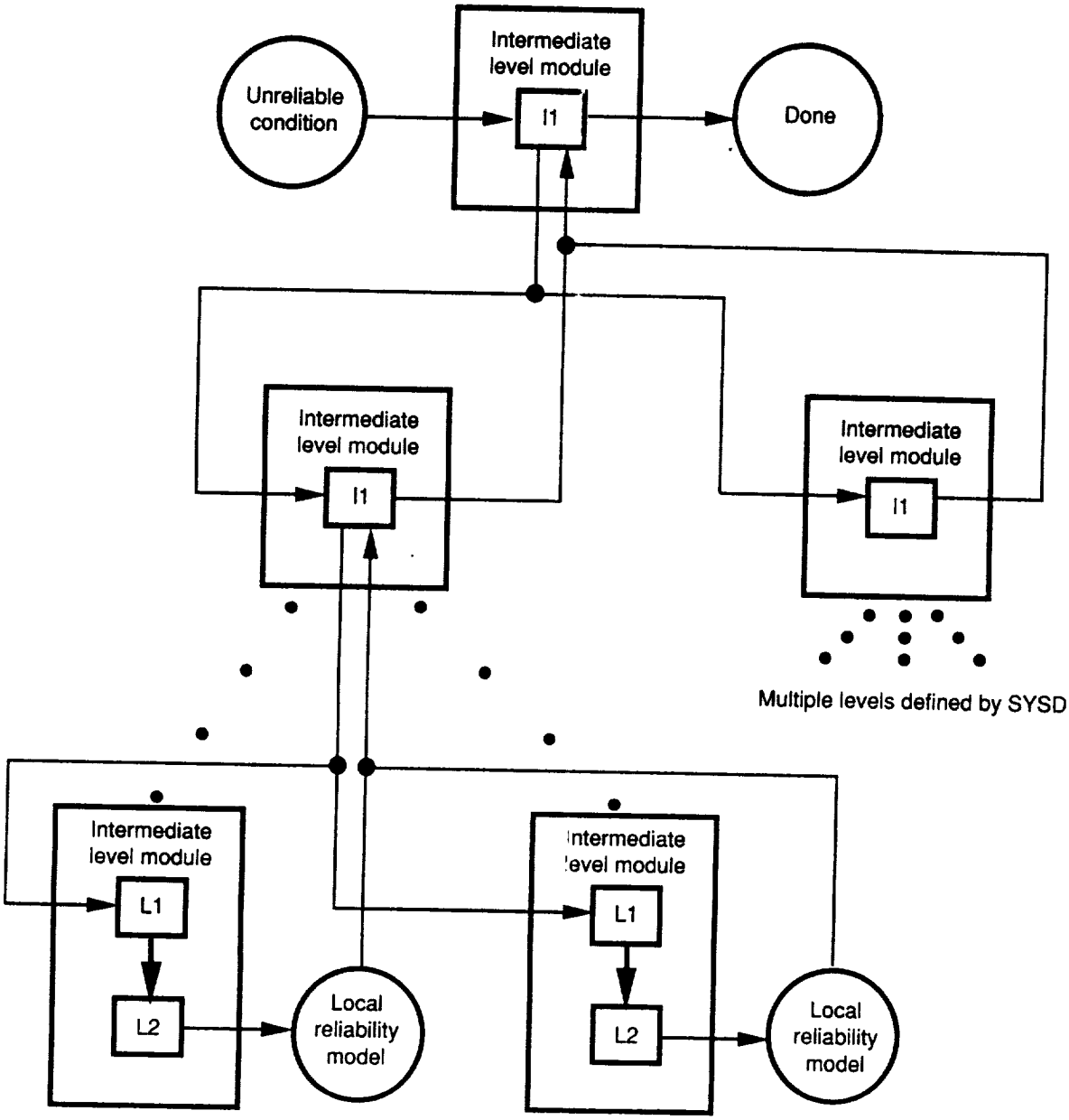


Figure 22. Local Model Generator

Initially, the highest level component module (I1) is instantiated. This process is involved primarily with controlling the order of invocation of the other LMG modules. Given a unreliable condition on the system, process I1 identifies the immediate subcomponents involved in the function. Separate reasoning modules for each subcomponent are then instantiated by process I1 in an order according to the functional flow.

If the first subcomponent in the function is a parent component also (i.e., has subcomponents defined in the BBD), another invocation of process I1 is invoked for it to identify the next lower level subcomponents involved in the function.

If a subcomponent is defined at the lowest level in the BBD where component failure modes are modeled, process L1 is invoked. Recall that failure modes are modeled as a change to the function, such that at the lowest level, there is a functional definition for the component under normal operating conditions and a functional definition for each possible failure mode. At this level where component failures are modeled, process L1 inputs a set of possible input characteristics received from its parent component. For each operational state of the component (non faulted and each failure mode), L1 traces the effects of all input characteristics through the function to determine the resulting output characteristics. L1 produces, for each state of the component, a set of OCDs and transitions for each output

characteristic that could possibly result from the component state and input conditions.

After all component states have been analyzed, Process L2 combines OCDs for the component states according to output characteristics so that a single set of OCDs for the component is defined. These OCDs together with the transitions comprise the component's local reliability model which is returned to process I1 of the parent component.

Process I1 passes the output characteristics to the subcomponent module which inputs these characteristics. If that subcomponent is defined in the BBD at an intermediate level, then I1 is invoked for it. Otherwise, L1 is invoked and a local reliability model is defined. This depth-first trace proceeds through the hierarchy until eventually, the outputs of the system are reached.

The resulting local reliability models for each of the lowest level components can be interrogated by the user or given to the RMAS processes discussed in the previous section in order to define a global model for the system. Several implementation details have been omitted in the general overview just presented. The next sections discuss each process in detail.

4.4.2. PROCESS I1

As stated above, Process I1, for each intermediate level component, manages the trace of the functional flow

between its subcomponents. First I1 invokes the module for the subcomponent who receives the parent's inputs. The module invoked is either an intermediate level module (I1) or a lowest level module (L1 and L2) depending on the level at which the corresponding subcomponent is defined in the BBD. Both modules return a list of OCDs for each subcomponents' outputs. Process I1 determines the next subcomponent to receive these output characteristics and determines the order of invocation of those subcomponents. The next subcomponent module is then invoked and given the possible input characteristics to analyze. When the output characteristics of this parent's outputs have been returned by the last subcomponent, Process I1 passes these characteristics to the parent module, I1, from which it was invoked. ,PP If the subcomponents interact in a cyclic manner within a parent module, process I1 must keep track of the input characteristics given to a subcomponent module so that when no new input characteristics are to be analyzed, the trace ends.

4.4.3. PROCESS L1

At the lowest level, where component failures are modeled, process L1 inputs a set of input characteristics received from its parent component. For each operational state of the component (non faulted and faulted) L1 determines the effects of the erroneous input characteristics on the component output characteristics. Thus, a given set of

C-2

input characteristics is compared against each failure mode for the system, and the output effects for all combinations are generated.

The next section introduces the algorithm for process L1 through an example function. Afterwards, a detailed discussion of several aspects of the algorithm is given.

The following example will be used to illustrate the steps of process L1:

L1 EXAMPLE

Consider a component that performs the following function:

```
[1]  input x1, x2, r;
     IF r = 0 THEN z = x1 + 3;
     If r = 1 THEN z = x2 + 4;
     output y = z   IF x1 > x2
            y = x2  IF x1 <= x2;
```

Using the nil sensitive operation default (discussed in section 3.1.2.2), this function would be encoded as:

```
[2]  input x1, x2, r;
     z = NS(x1)   | r = 0      (* z = x1 + 3; *)
         NS(x2)   | r = 1;     (* z = x2 + 4; *)
     output y = z   | x1 > x2
                x2 | x1 <= x2;
```

Consider input characteristics: x1:(g,b,n), x2:(g,b) and r = 0,1. Process L1 defines the possible output characteristics (g,b,n) given this function and the possible input characteristics. The output characteristic definition that is produced by L1 for this function and the input characteristics is:

[3] OUTPUT CHARACTERISTIC DEFINITION FOR Y:

output g IF x1:(g) and x2:(g) and r=1
OR x1:(b) and x2:(g) and r=1
OR x1:(g) and x2:(g)

output b IF x1:(b) and r=0
OR x1:(b) and x2:(b)
OR x1:(g) and x2:(b)

output n IF x1:(n)

If any failure modes are to be modeled for this component, then the additional function definitions for each failure mode would have been defined in the BBD, and these would be analyzed in the same manner by process L1. For this illustration, assume that no failure modes are modeled, and therefore, the OCD in [3] is the only output of process L1.

The algorithm shell for process L1 is shown in figure 23. Process L1 has a nested looping structure that iterates over all sentences within a function, and all clauses within a sentence. Three phases are distinguishable.

```
L1:  For each function:
Phase 1:    For each sentence (delineated by ;)
                  For each clause (delineated by |)
                        Substitute characteristics in for input variables
                        Do until (no more substitutions)
                           Propagate
                           Substitute
                        End do;
Phase 2:                Case (type to left of |)
                              1: function ...
                              2: variable ...
                              3. characteristic ...
                        end case;
                  end; (for each clause)
Phase 3:          Combine clauses according to output characteristic
                  Detect and correct overlaps between OCDs
                  Detect and correct overlaps within OCDs
                  Separate condition from data characteristics
                  Remove intermediate variable characteristics
            end; (for each sentence)
      end; (for each function)
end; (L1)
```

Figure 23. Skeleton for Process L1

In the first phase, the characteristics for each variable are substituted into a given clause. Initially, only input characteristics are known. These characteristics are then propagated through the conditions and functions defined in the clause. Each propagation results in a generation of characteristics for other variables. These new characteristics are substituted in for all other occurrences of the variable, propagated and so forth until no new substitutions may be made. The manner and order of this propagation/substitution phase will be illustrated generally for the example above, and then explained in sections 3.4.3.2.1 At the end of phase one, the possible output characteristics of a given clause are known.

In the second phase, each the output of the clause is instantiated for each possible output characteristic. This substitution triggers a reiteration of propagation/substitutions until each output characteristic is defined in terms of input characteristics. The definition of each output characteristic as a relationship to the input characteristics is known as the output characteristic definition (OCD).

After all clauses for a sentence have been analyzed, phase three combines OCDs for a common output characteristic. These OCDs are then modified so that:

1. The resulting OCDs for a sentence do not overlap between characteristics. Overlaps might lead to an

erroneous model for the component. For example:

output g if x:(g)

output b if x:(g) OR y:(b)

is not acceptable because the output can not be both g and b when x:(g). Therefore, phase 3 ensures that no overlaps occur between OCDs.

2. Overlaps within a particular output characteristic should also be eliminated if possible. For example:

output g IF x:(g) OR

IF x:(g) AND y:(b)

could be simplified to:

output g if x:(g)

3. Conditions such as x1 > x2 or x1 <= x2 are not included in the OCD, since the values of x1 or x2 that determine the truth or falsity of the condition are not known. Rather, the condition is separated from the variable characteristics in the OCD.

4. The output characteristic definitions must be defined in terms of the input characteristics without reference to intermediate variables. For example, the OCDs in [3] do not include reference to variable z since z is an intermediate variable dependent on the inputs x1 and x2. Therefore, after the conditions are separated from the characteristics, the characteristics of intermediate variables are removed from the OCD (but the

variables remain in the conditions).

5. For each clause within the resulting OCD, any variable whose characteristic set includes all possible characteristics (g,b,n) can be eliminated from the clause since the characteristic of that variable is not a determinant of the characteristic output for the clause.

4.4.3.1. EXAMPLE OF PROCESS L1

The next sections describes the three phases of process L1 for the example shown above. The correlation to the exact phases and steps of the algorithm will not be maintained for this illustration. For instance, for the sake of brevity, two clauses within a sentence are often analyzes concurrently, even though the algorithm analyzed each sequentially. Rather, the intent of this example is to give the reader an understanding of the operations of substitutions, propagations, and the general goal of process L1. The sections following this example delve into the underlying mechanics of the process. The algorithm itself can be found in Appendix B.

PHASE 1:

For each sentence of the function (delineated by ;)

1. Substitute all input characteristics in for input variables.

2. Propagate the input characteristics through the conditions and operations of the function until the output characteristics are defined.

FIRST SENTENCE:

```
[4]  z = NS(x1)   | r = 0      (* z = x1 + 3 *)
         NS(x2)   | r = 1;     (* z = x2 + 4 *)
```

1. Substitute x1:(g,b,n) and x2:(g,b) in for variables x1 and x2 respectively:

```
z = NS(x1:(g,b,n))   | r = 0     (* clause 1 *)
    NS(x2:(g,b))     | r = 1;    (* clause 2 *)
```

2. Propagate the input characteristics of x1 and x2 through all conditions and functions in the clause to the right of the | symbol (each clause is delineated by |).

Not applicable in this example.

PHASE 2:

1. Instantiate the characteristics of each clause's output into separate clauses.

The output of these clauses are the result of some operation, NS. Recall that nil sensitive functions (NS) will output nothing (or nil) if any of the inputs is nil. Therefore, the output characteristic possibilities for NS functions is:

NS OUTPUT = good IF all operands are good
nil IF any operand is nil
bad IF any operand is bad
AND no operand is nil

Since there is only one operand in each of the NS operations above, applying the NS rules to [4] results in:

z:(g) | x1:(g) and r=0
z:(b) | x1:(b) and r=0
z:(n) | x1:(n) and r=0

for the first clause, and:

z:(g) | x2:(g) and r=1
z:(b) | x2:(b) and r=1

for the second clause. [2]

PHASE 3:

1. Combine the two clauses according to common characteristics for z:

[5] z:(g) | x1:(g) and r=0
| x2:(g) and r=1
z:(b) | x1:(b) and r=0
| x2:(b) and r=1

[2] Variables not referenced in a clause are assumed to have full characteristic possibilities (i.e., the characteristic of these variables is irrelevant to the outcome of the clause.

z:(n) | x1:(n) and r=0

Phase 3 also changes the OCDs to eliminate overlapping conditions among clauses. However, there are no such overlaps in these OCDs, and therefore, a discussion of overlaps is deferred until analysis of the next sentence.

SECOND SENTENCE:

PHASE 1:

Referring to the second sentence in [2]:

```
[6]  output y = z   | x1 > x2
            y = x2  | x1 <= x2;
```

The second sentence is analyzed in the same way using the characteristics for z:(g,b,n) defined in [5] and the input characteristics x1:(g,b,n) and x2:(g,b). However, not all combinations of input characteristics and characteristics of z are possible. For example, z can not have a characteristic of b if both x1 and x2 are g. To detect these inconsistencies, the intermediate variable is substituted in for each characteristic and the input conditions under which the z obtains its characteristic is included in the condition for the clause. For example, the substitution of z:(g) in for z must be accompanied by the possible conditions of the input characteristics that enable z:(g) to hold. In this way, separate clauses for each possible substitution are defined:

Consider each possible combination of z, x1 and x2 for the first clause in sentence [6]:

[6] output y = z | x1 > x2;

PHASE 1:

1. For z:(g) and x1:(g,b,n) and x2:(g,b) and r=(0,1):

```
[7]  output g | x1:(g,b,n) > x2:(g,b)
                {and x1:(g) and r=0} and z:(g)
              | x1:(g,b,n) > x2:(g,b)
                {and x2:(g) and r=1} and z:(g)
```

The conditions within {} indicates conditions that must hold for z:(g) to be true (see [5]).

First, find the intersection of characteristic possibilities for all variables in each clause:

x1: (x1:(g,b,n) and x1:(g) --> x1:(g)) in the first clause of [7]

x2: (x2:(g,b) and x2:(g) --> x2:(g)) in the second clause of [7]

The result after this substitution is:

```
output g | x1:(g) > x2:(g,b) and r=0 and z:(g)
         | x1:(g,b,n) > x2:(g) and r=1 and z:(g)
```

Next, note that x1 is involved in a ">" comparison in the second clause and has a characteristic possibility of

"n". It is meaningless to compare a variable that is "n". The condition is assumed to fail. Therefore, the "n" characteristic is removed from the set for x1. To ensure that the final OCD is complete for all input characteristics, an additional clause is created that specifies that the output is "n" if any input is "n":

[8] output g | x1:(g) > x2:(g,b) and r=0 and z:(g)
. | x1:(g,b) > x2:(g) and r=1 and z:(g)
output n | x1:(n)

PHASE 2 (Instantiate the clauses for each characteristic):

Since the clauses already specify a single output characteristic, no further instantiations need be done.

PHASE 1:

2. For z:(b) and x1:(g,b,n) and x2:(g,b):

[9] output b | x1:(g,b,n) > x2:(g,b)
{and x1:(b) and r=0} and z:(b)
| x1:(g,b,n) > x2:(g,b) {and x2:(b) and r=1} and z:(b)

Simplify [9] by taking the intersection of characteristics for each variable:

output b | x1:(b) > x2:(g,b) and r=0 and z:(b)
| x1:(g,b,n) > x2:(b) and r=1 and z:(b)

Here, again the "n" clause can be eliminated from the

characteristic possibilities for x1 and an additional clause can be created.

[10] output b | x1:(b) > x2:(g,b) and r=0 and z:(b)
| x1:(g,b) > x2:(b) and r=1 and z:(b)
output n | x1:(n)

PHASE 2 (Instantiate the clauses for each characteristic):

Since the clause already specifies a single output characteristic, no further substitutions need be done.

PHASE 1:

3. For z:(n) and x1(g,b,n) and x2:(g,b):

[11] output n | x1:(g,b,n) > x2:(g,b) {and x1:(n) and r=0} and z:(n)

Simplify:

output n | x1:(n) > x2:(g,b) and r=0 and z:(n)

Again, the "n" clause can be eliminated from the characteristic possibilities for x1 and an additional clause can be created. However, this leaves the original clause with no possible characteristics for x1. Therefore, the clause is invalid and can be eliminated altogether (The new clause remains).

[12] output n | x1:(n)

PHASE 2 (Instantiate the clauses for each characteristic):

Since the clause already specifies a single output characteristic, no further substitutions need be done.

Summary for the first clause of the second sentence (combine [8], [10], and [12])

```
[13] output g | x1:(g) > x2:(g,b) and r=0 and z:(g)
              | x1:(g,b) > x2:(g) and r=1 and z:(g)
     output b | x1:(b) > x2:(g,b) and r=0  and z:(b)
              | x1:(g,b) > x2:(b) and r=1 and z:(b)
     output n | x1:(n)
```

Second clause of sentence in [6]:

```
     output y = x2 | x1 <= x2;
```

PHASE 1:

1. For x2:(g) and x1:(g,b,n):

```
[14] output g | x1:(g,b,n)  <= x2:(g);
```

AFTER PHASE 2:

```
[15] output g | x1:(g,b) <= x2:(g)
     output n | x1:(n)
```

PHASE 1:

2. For x2:(b) and x1:(g,b,n):

```
[16] output b | x1:(g,b,n)  <= x2:(b);
```

AFTER PHASE 2:

[17] output b | x1:(g,b) <= x2:(b)

output n | x1:(n)

Summary for the second clause of the second sentence: (combination of [15] and [17])

[18] output g | x1:(g,b) <= x2:(g)

output b | x1:(g,b) <= x2:(b);

output n | x1:n

PHASE 3:

Combine the characteristics for the output for the two clauses of the second sentence ([13] and [18]):

[19] OUTPUT CHARACTERISTIC DEFINITION FOR Y:

output g | x1:(g) > x2:(g,b) and r=0 and z:(g)

| x1:(g,b) > x2:(g) and r=1 and z:(g)

| x1:(g,b) <= x2:(g)

output b | x1:(b) > x2:(g,b) and r=0 and z:(g)

| x1:(g,b) > x2:(b) and r=1 and z:(g)

| x1:(g,b) <= x2:(b)

output n | x1:(n)

As stated in the general overview, phase 3 is responsible for ensuring that the newly created OCD for a component is complete and non-redundant: The OCD is complete if it specifies and output characteristic for every possible input characteristic, and it is non-redundant if there is no input

characteristic combination for which more than one output characteristic is defined. Also, characteristics for intermediate level variables for the sentence (i.e., variables that are not inputs to the function or outputs for the sentence) are eliminated from the OCD:

[20] OUTPUT CHARACTERISTIC DEFINITION FOR Y:

```
output g | x1:(g) > x2:(g,b) and r=0      [A]
         | x1:(g,b) > x2:(g) and r=1       [B]
         | x1:(g,b) <= x2:(g)              [C]
output b | x1:(b) > x2:(g,b) and r=0       [D]
         | x1:(g,b) > x2:(b) and r=1       [E]
         | x1:(g,b) <= x2:(b)              [F]
output n | x1:(n)                         [G]
```

Phase 3 must analyze the OCD and detect any overlaps in input characteristics. Overlaps occur because the evaluation of a function is dependent on conditions among input and internal variables. For example, the output of a function may be dependent on a equality comparison of two variables. The determination of the truth or falsity of the conditions is dependent on the value of the data. However, the value of the data is not known. Rather, only the characteristics of the data is known. This is not so much an incomplete specification of functions as it is a symptom of the incomplete description of faults in general. For example, when a component fails, and it is said that the failure causes corrupted output. The value of the corrupted output is not known. Another component acting on the corrupted input may

test the input in a condition. The result of the condition can not be known since the value is not known.

The results of this is a non-deterministic output characteristic specification in which two or more output characteristics may be defined on a particular combination of input characteristics, the distinguishing factor being the internal condition.

Overlaps can take on three types:

1. the input characteristics can be identical and the difference is soley in the conditions

   (e.g. x:g and y:g x < y
   x:g and y:g x >= y)

2. the input characteristics of one domain can be a subset of the input characteristics of the other domain, so that there is some characteristic that is in one domain and not the other

   (e.g. x:g and y:g,b x < y
   x:g and y:g x < y)

3. Or the overlap may be a combination of the two:

   (e.g. x:g and y:g,b x < y
   x:g and y:g x >= y)

Further, the overlap can occur within or between output characteristics. It is advantageous to resolve overlaps

between characteristics first so that the resulting output definition for the component would be complete. Afterwards, overlaps among clauses within an output characteristic definition may be eliminated so that a more concise OCD results.

For dealing with overlaps between OCDs, the following options are ranked in order of preference:

1. Heuristics may be applied depending on the output characteristics involved in the overlap. These heuristics are patterned after the way such overlaps are handled manually. For example,

   a. If the overlap is between a good characteristic and a bad characteristic, and if one of the overlapping characteristics is a bad characteristic, then a conservative assumption would shift the overlap to the bad output characteristic.

   b. If the overlap is between a nil characteristic and a good or bad characteristic, and if one of the overlapping characteristics is a nil characteristic, then a reasonable assumption would shift the overlap to the nil characteristic.

2. It is possible that some of the characteristic possibilities in the condition are not possible. For example, a reasonable assumption would conclude that a variable, x1, whose value is good, (i.e., x1:(g)) would not be equivalent

to a variable, x2, whose value is corrupted (i.e., x2:(b)). Therefore, a clause containing the condition, x:g = y:(g,b), can be reduced to x:g = y:b by assuming that a corrupted input will not equal a non corrupted input. This is a reasonable assumption that may be used to eliminate characteristic possibilities from clauses. Other assumptions that may hold are:

x:b = y:b

x:g = y:g

However, the assumption may not always hold. Therefore, the analyst is interrogated as to the validity of the assumption if it is relevant to the particular clause. Because these particular assumptions would apply to most cases, the user is interrogated on these assumptions (called condition propagation rules) during phase one. In phase three, when the process L1 detects an overlap between two clauses in different OCDs, the user is asked if the intersecting characteristics may be applied to one or the other OCD.

3. Failing a resolution of the overlap by options 1 or 2, the user may elect to define a non deterministic output characteristic definition, by including for each path, and estimate of the probability of occurrence of the distinguishing characteristic or condition. However, the use of this option is not encouraged because of the percentage assigned to each possibility is often subjective (whereas

the other options have some logical foundation) and the resulting OCD has more conditions to analyze.

Returning to the example in [20], the following overlaps between output characteristics are detected.

Overlaps:

[A] and [F]

[C] and [D]

[A] and [F]: overlap on x1:(g) and x2:(b) and r=0

Option 1a applies to this set of overlapping clauses, and therefore the user would be interrogated as follows:

Query: Given the following overlapping clauses:

output g | x1:(g) > x2:(g,b) and r=0 [A]

output b | x1:(g,b) <= x2:(b) [F]

(the overlapping condition is: x1:(g) and x2:(b) and r=0) Can the overlapping condition be assigned to the OCD for characteristic "b"?

The user would answer yes, and process L1 would subtract the intersecting characteristics from the OCD for g:

Result:

output g | x1:(g) > x2:(g) and r=0 [A]

output b | x1:(g,b) <= x2:(b) [F]

[C] and [D]: overlap on x1:(b) and x2:(g)

Option 1a applies to this set of overlapping clauses

also, and after an affirmative response to a user query, the result would be:

```
Result:
     output g | x1:(g) <= x2:(g)              [C1]
              | x1:(b) <= x2:(g) and r=1      [C2]
     output b | x1:(b) > x2:(g,b) and r=0     [D]
```

RESULT AFTER OVERLAPS BETWEEN OCDs HAVE BEEN RESOLVED:

```
     output g | x1:(g) > x2:(g) and r=0       [A]
              | x1:(g,b) > x2:(g) and r=1     [B],[C2]
              | x1:(g) <= x2:(g)              [C1]

     output b | x1:(b) > x2:(g,b) and r=0     [D]
              | x1:(g,b) > x2:(b) and r=1     [E]
              | x1:(g,b) <= x2:(b)            [F]

     output n | x1:(n)                        [G]
```

Once overlaps between output characteristic definitions have been eliminated, phase 3 analyzes each OCD and detects overlaps between clauses within an OCD. It is not as critical that these overlaps be eliminated since it is only the overlaps between OCDs that make the resulting model ambiguous or incomplete. However, phase 3 detects clauses within an OCD that are subsets of other clauses and eliminates that subset clause from the final OCD. This reduces the size of the final OCD.

Referring to the OCD in [20], the following overlaps

within output characteristic definitions are identified:

1. [A] AND [C1]

2. [B] AND [C1]

3. [D] AND [F]

4. [E] AND [F]

The first and fourth overlaps above are super/subset clauses. Therefore, clause [A] and [E] may be eliminated. Combining these, the resulting OCD is:

RESULT AFTER OVERLAPS WITHIN HAVE BEEN RESOLVED

```
output g | x1:(g,b) > x2:(g) and r=1      [B]
         | x1:(g) > x2:(g)                 [C1]

output b | x1:(b) > x2:(g,b) and r=0      [D]
         | x1:(g,b) <= x2:(b)              [F]

output n | x1:(n)                         [G]
```

Finally, to produce the final output characteristic definition, the conditions are separated from the characteristics. Also, any variables in clauses that involve an enumeration of all possible data characteristics may be eliminated from the clause.

SEPARATE CONDITION FROM INPUT CHARACTERISTIC:

```
output g | x1:(g,b) and x2:(g) and r=1   (x1 > x2)   [B]
         | x1:(g)   and   x2:(g)              (x1  >  x2)
```

[C1]

output b | x1:(b) and x2:(g,b) and r=0 (x1 > x2) [D]

| x1:(g,b) and x2:(b) (x1 <= x2) [F]

output n | x1:(n) (x1 > x2) [G]

ELIMINATE FULLY ENUMERATED VARIABLES:

There are no variables in this OCD that have fully enumerated characteristic sets.

With the clause delineator "|" replaced by the logical OR, this OCD is identical to the one presented in [3].

Note that the final OCD for this component is not fully defined. For example, it does not specify the output characteristic for all cases that x2:(n). However, it is defined on all input characteristics given to L1. This is one way in which the model created by the LMG may be superior to a user defined model on all inputs in that there are possibly less conditions for the RMAS to consider. The model will not be incorrect in the RMAS if an impossible combination is modeled in the OCD. However, it would be erroneous if a combination that is possible is not modeled. Therefore, by modeling all combinations (as may be done if the user enters the local reliability models for the RMAS), the state space may be larger than necessary, but the model would not be less correct.

4.4.3.2. SPECIAL NOTES

4.4.3.2.1. Phase 1

As shown in the example, the first phase is responsible for propagating all input characteristics and local variable characteristics through the current sentence until output characteristics are defined. This propagation is accomplished through an iteration of:

a. Substituting characteristics in for variables, and

b. Propagating these characteristics through functions and conditions in which the variables are involved in order to define characteristics for other variables.

Three special attributes of the substitution/propagation phase merit discussion:

1. Define the set of characteristics possibilities for a variable at an instance in the analysis as the variable's characteristic set. As a result of propagating characteristics through an operation or a function, other variables' characteristic sets may be changed. These changes must be substituted in for all instances of these variables and propagated through any other affected functions. Therefore, at any one time in phase 1, there will be many variables whose characteristics must be substituted/propagated. In order to perform this in an organized manner, a given substitution/propagation is performed until all variables

that are affected have had their new characteristic sets substituted/propagated. This is a depth-first approach.

2. Intermediate variable characteristic possibilities are dependent on other variable characteristic sets. For example, an intermediate variable may have a characteristic of "g" only if an input variable's characteristic is "g" according to analysis of a previous sentence in the function. When the intermediate variable is characteristics are substituted/propagated, these dependencies must be substituted in also.

3. Further, for a given set of variables to be substituted at any one time, a priority ordering that favors intermediate variable substitutions before input characteristic substitutions is used. This reduces the number of substitutions that must be made since (as explained above) conditions upon which the intermediate variable characteristics depend are substituted in with these characteristics.

Substitution:

Substitution is performed as follows: Given a set of characteristics to be substituted in for a variable, the intersection of these characteristics with the characteristics currently associated with the variable is calculated. This intersection is then considered to be the new set of characteristics for the variable. Each instance of the variable in the clause is then updated with the new set of characteristics.

Propagation:

Propagation consists of the following: For each instance of the variable, the the variable characteristics are propagated through the operation or condition in which the variable is involved. These conditions and operations may be nil-sensitive operations, non-nil sensitive operations, or boolean comparators such as =, <>. Other boolean comparators such as <, >, <=, >=, are not propagated through since the result of the boolean comparison is dependent on the value of the variables involved, and these values are not generally known. For example, what is the result of the comparison x > y when x has the characteristic of "b"? The result is dependent on the value of x, and that value is not known. Therefore, assumptions regarding boolean comparators are handled interactively with the user in phase 3. The following details the propagation rules for nil-sensitive and non-nil-sensitive operations and equality comparators.

A. Nil-sensitive operations:

If the variable is a parameter in a nil sensitive operation, and if all parameters of the operation are characterized (i.e., all have characteristics currently associated with them), then the nil sensitive propagation rules are applied to separate the clause into three types of clauses (i.e., for "g,b,n" characteristic outputs) according to the parameters' characteristics:

Nil-sensitive propagation rules:

1. "G" clause:

Condition:

If all parameters have a characteristic possibility of "g", then:

Action:

Create a clause in which the result of the function is assigned characteristic "g" and all parameters for this clause are assigned characteristic "g".

2. "N" clauses:

Condition:

If there is any parameter with a "n" characteristic, then:

Action:

Create a clause for EACH variable x(i) that has a "n" characteristic in which the result of the function is assigned characteristic "n", the variable x(i) is assigned the characteristic "n", and assign all other variables, x(k) where k > i, (according to some predetermined ordering) retain their currently assigned characteristics (i.e., no changes to these variable characteristics). Further, remove from variables, x(j) where j < i, the characteristic "n".[3] If any resulting variable sets are empty (i.e., the only characteristic was "n" for that set), then remove the clause.

3. "B" clauses:

Condition:

If there are no parameters that have "n" as the only characteristic possibility and there is any parameter with a "b" characteristic, then:

Action:

Create a clause for EACH variable x(i) that has a "b" characteristic possibility if all variables x(j), where j < i, have a characteristic possibility "g".[4] The new clause is formed by:

a. assigning the variable x(i) a "b" characteristic

b. the result of the function is assigned characteristic "b",

c. all variables x(j) for j < i are assigned the "g" characteristic,

d. assign all other variables, x(k) where k > i, (according to some predetermined ordering) retain their currently assigned characteristic sets with

[3] Note: Since no two clauses may address intersecting conditions (see section 3.1.2.2), the clauses, must define a exclusive OR condition, instead of simply a boolean inclusive OR. The last change to the transitions for x(j), where j < i satisfies this.

[4] This additional condition also ensures that the clauses define a exclusive OR (see previous footnote). However, removing "b" from the previous variables is not sufficient since the characteristic of "n" is not allowed either. Therefore, all variables prior to this variable must have a "g" characteristic.

"n" removed.[5]

A flowchart for Nil-sensitive propagation is shown in figures 24 and 25.

B. Non-nil-sensitive operations:

If the variable is a parameter in a non-nil sensitive operation, and if all parameters of the operation are characterized (i.e., all have characteristics currently associated with them), then the non-nil sensitive propagation rules are applied to separate the clause into three types of clauses (i.e., for "g,b,n" characteristic outputs of the function) according to the parameters' characteristics:

Non-nil-sensitive propagation rules:

1. "G" clause:

Condition:

If all parameters have a characteristic possibility of "g", then:

Action:

Create a clause in which the result of the function is assigned characteristic "g" and all parameters for this

[5] The number of clauses created by this algorithm would be reduced if an a priori sorting of the variable parameters would place those variables without "g" characteristics ahead of those variable characteristics with "g".

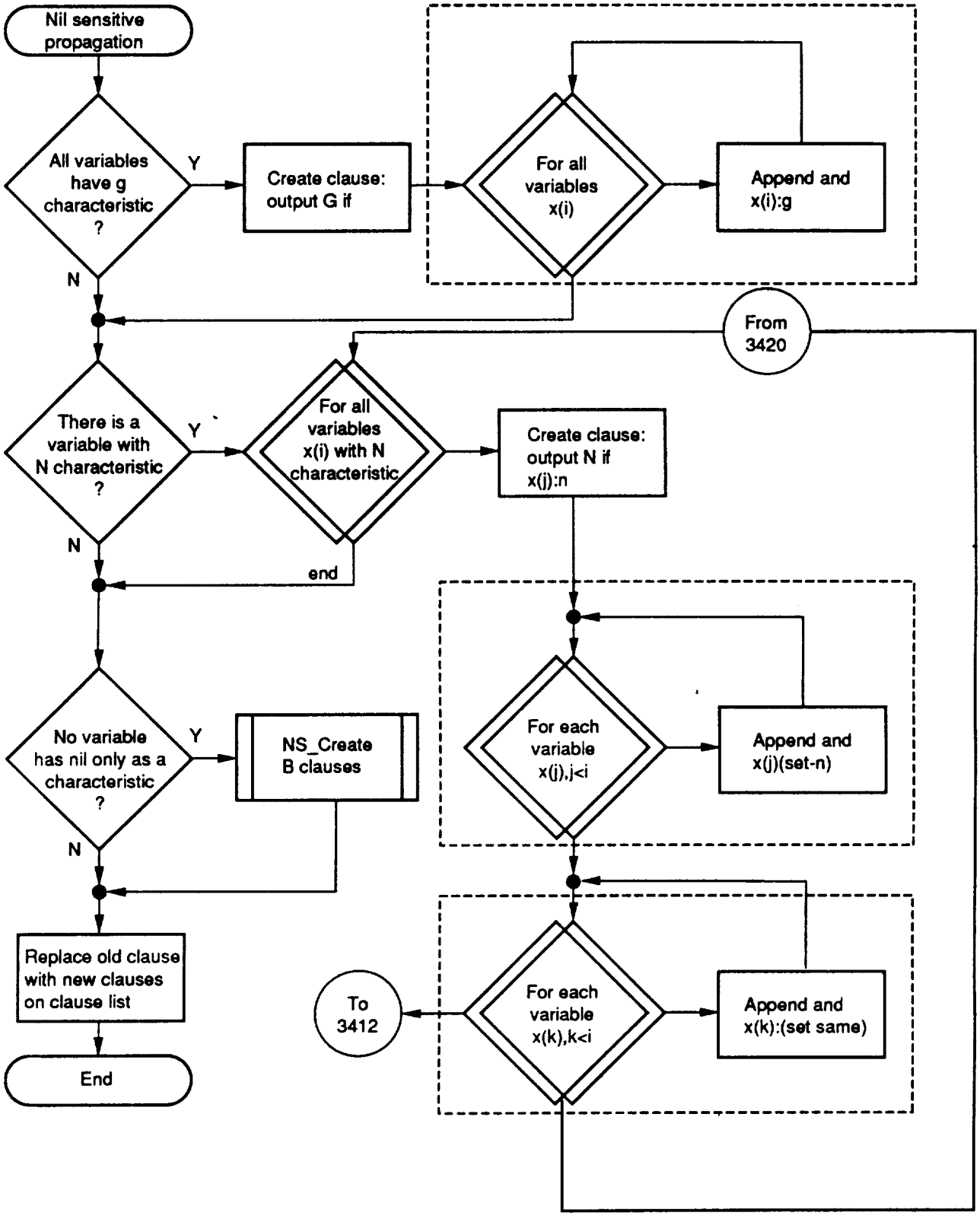


Figure 24. Nil Sensitive Propagation

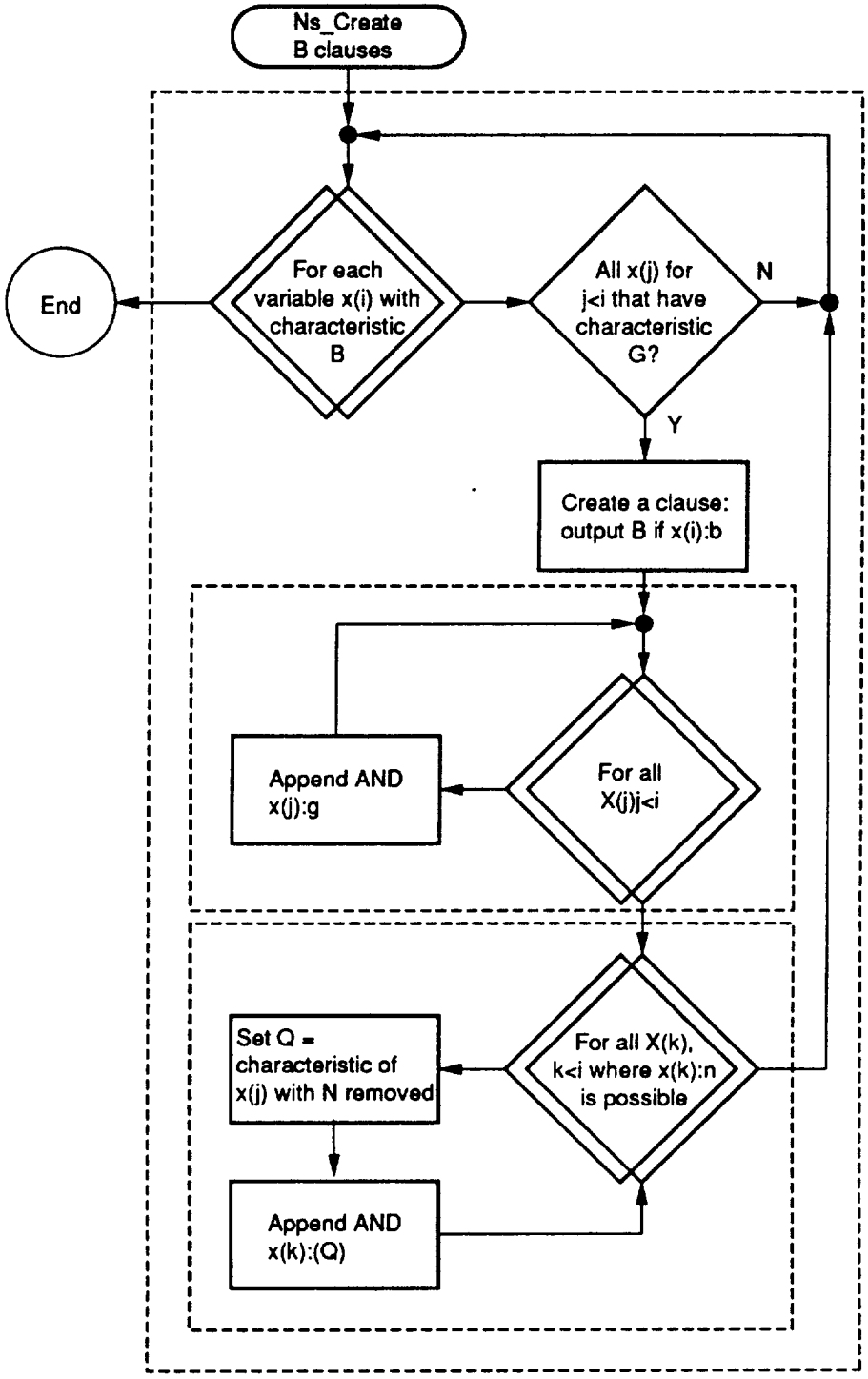

Ns_Create
B clauses
End
For each
variable x(i) with
characteristic
B
All x(j) for
j<i that have
characteristic
G?
N
Y
Create a clause:
output B if x(i):b
Append AND
x(j):g
For all
X(j)j<i
Set Q =
characteristic of
x(j) with N removed
For all X(k),
k<i where x(k):n
is possible
Append AND
x(k):(Q)

Figure 25.

clause are assigned characteristic "g".

2. "N" clauses:

Condition:

If all parameters have a characteristic possibility of "n, then:

Action:

Create a clause in which the result of the function is assigned characteristic "n" and all parameters for this clause are assigned characteristic "n".

3. "B" clauses:

Condition:

There is any parameter with a "b" characteristic, then:

Action:

Create a clause for EACH variable x(i) that has a "b" characteristic in which the result of the function is assigned characteristic "b", the variable x(i) is assigned the characteristic "b", and assign all other variables, x(k) where k > i, (according to some predetermined ordering) retain their currently assigned characteristics (i.e., no changes to these variable characteristics). Further, remove from variables, x(j) where j < i, the characteristic "b".[6] If any resulting variable sets are empty (i.e., the only characteristic was "n" for that set),then remove the clause.

A flowchart for Non-nil-sensitive propagation is shown in figure 26 and 27.

C. Condition Propagation Rules:

There are three types of assumptions which may apply to boolean comparators for equality (=) and inequality (<>). These are:

1. x:g <> y:b assumption: Assume that a variable that is "g" is never equal to a variable that is "b". In other words, a condition "x:g = y:b" always fails. Again, the equality of two variables can not be determined without knowledge the their values. However, most analysts assume that if a value is corrupted by some failure, that it would not be equal to another variable that was not corrupted. To verify that this assumption holds, the following rule is defined:

Condition:

equality condition in the clause (x:(set1) = y:(set2)) where set1 or set2 has "b" and the other set has "g".

Query: Can it be assumed that x:b <> y:g and x:g <> y:b?

Action:

Create 4 separate clauses:

[6] Note: Since no two clauses may address intersecting conditions (see section 3.1.2.2), the clauses, must define a exclusive OR condition, instead of simply a boolean inclusive OR. The last change to the transitions for x(j), where j < i satisfies this.

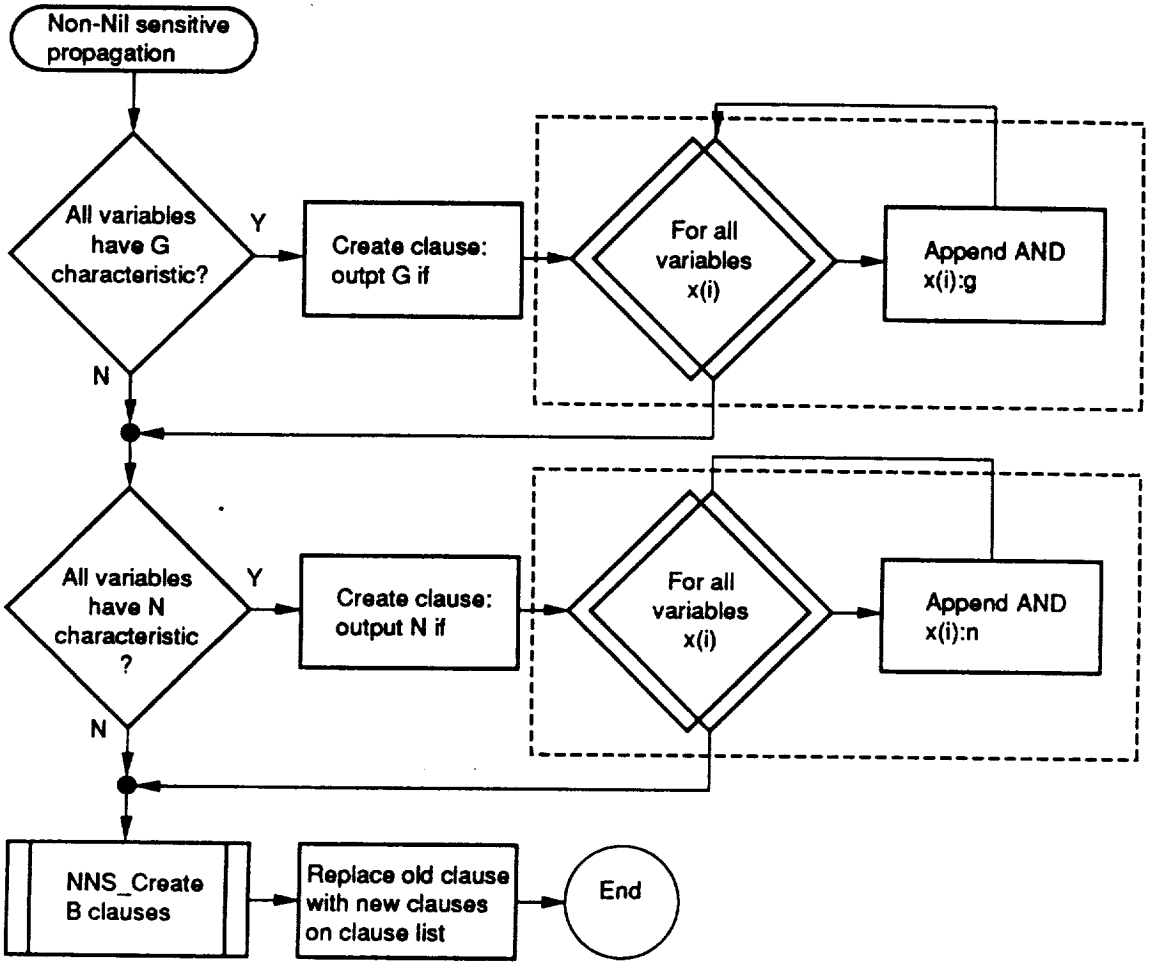


Figure 26. Non-Nil Sensitive Propagation

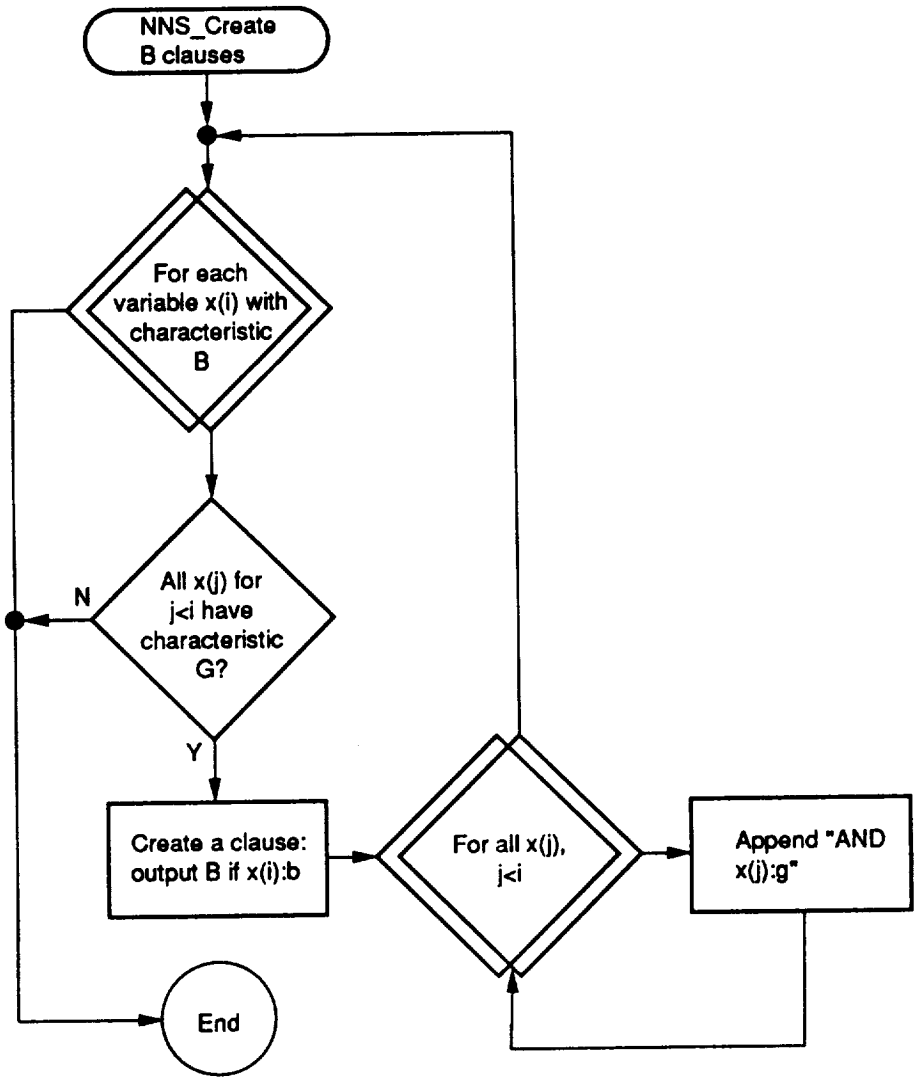


Figure 27. Non-Nil Sensitive Propagtion (cont.)

1. if "g" in set1 then remove "b" from set2

2. if "b" in set1 then remove "g" from set2

3. if "g" in set2 then remove "b" from set1

4. if "b" in set2 then remove "g" from set1

Eliminate duplicate clauses and clauses with set1 or set2 = 0.

2. x:g = y:g assumption: This assumption is common for comparing variables that relate to the same redundant component type. It assumes that in comparing two variables that are both "g" (i.e., not corrupted), the equality test will hold. To verify that this assumption holds, the following rule is defined:

Condition:

Inequality Operation (x:(set1) <> y:(set2)) where set1 and set2 have "g"

Query: Can it be assumed that x:g = y:g?

Action:

Create 2 clauses:

1. if "g" in set1 then remove "g" from set2

2. if "g" in set2 then remove "g" from set1

Eliminate duplicate clauses and clauses with set1 or set2 = 0.

3. x:b = y:b assumption: This assumption is often used as a simplifying assumption to handle a worst case analysis. For example, the analysis of a voter component that outputs the majority of the inputs may assume a worst case scenario in which all corrupted ("b") inputs have that same value, and therefore, may outvote a "g" value. To verify that this assumption holds, the following rule is defined:

Condition:

Inequality Operation ((x:(set1) <> y:(set2)) where set1 and set2 have "b":

Query: Can it be assumed that x:b = y:b?

Action:

1. if "b" in set1 then remove "b" from set2

2. if "b" in set1 then remove "b" from set2

Eliminate duplicate clauses and clauses with set1 or set2 = 0.

4. Nil comparison assumption: This assumption is the only one applied to non-equality boolean comparators. As illustrated in the previous example, it assumes that a comparison to a nil value always fails. The rule is defined as follows:

Condition:

Boolean comparators (=, <, >, <=, >=) where set1 or set2 contain an "n" characteristic:

Action

1. Remove "n" from the set1 and set2

2. Create a clause: N | {x:(n)} AND {y:(n)} where {} indicates inclusion only if the variable has a "n" characteristic.

These four assumptions have been identified as common assumptions made by analysts. If further assumptions are identified at a later time, interactive inquiries may be added to this list. Further reductions are applied in phase 3.

4.4.3.2.2. Phase 2:

After all the substitutions and propagations have been performed for the clause, the consequent part of the clause (to the left of the | in the current functional definition) is instantiated for each possible output characteristic. There are three types of consequents to a clause: a function, a variable, or a single characteristic.

1. Characteristic: If the consequent is a characteristic, then no instantiation is necessary.

2. Variable: If the consequent is a variable, then a separate clause is created for each characteristic associated with the variable. For each new clause, the characteristic is substituted into the characteristic set for that variable, and the substitution/propagation algorithm is

executed for the clause.

3. Function: If the consequent is a function, the appropriate function propagation rules are applied to the function (NS or NNS function propagation rules), and the substitution/propagation algorithm is executed for the clause.

4.4.3.2.3. Phase 3:

After all clauses have been instantiated for the sentence, the clauses are grouped, in phase 3, according to common output characteristic. Next, the algorithm analyzes each group pair to detect overlapping conditions between output characteristics. After all overlaps between groups have been eliminated, each group is analyzed to detect overlaps within an output characteristic definition so that a more concise OCD is defined. The previous example discussed the motivation for eliminating overlapping conditions and the heuristics used to resolve the conflicts. Therefore, such information is not repeated in this section. After all conflicts have been resolved within and between output characteristic groups, the input characteristics are separated from both the intermediate variable characteristics and the internal boolean conditions. This is because the final OCD for the subcomponent must define the output characteristics of the function in terms of the input characteristics and the state of the subcomponent. References to the state of the subcomponent are added in process

L2 after all failure and non-failure states have defined OCDs.

4.4.4. PROCESS L2

Process L2 is called after process L1 has analyzed each failure mode function and the non-failed function for a component. The purpose of process L2 is to combine the OCDs for each failure mode state into a single OCD for the component.

The algorithm for process L2 is straightforward. For each output characteristic, a combined OCD is created by concatenating the failure mode state variable and the OCD for the failure mode state into a single clause. Thus, the combined OCD is defined as:

```
OUTPUT (output characteristic)
   IF (OCD in L1) AND (fault state)
      OR {next fault state}
```

For example, recall the voter component of figure 8. If two failure modes BAD, and NO-OP were also modeled for the voter, process L1 would return the following OCDs for each failure mode:

```
OUTPUT y:b
IF VOTER_NOF TRANTO VOTER_BAD;

OUTPUT y:n
IF VOTER_NOF TRANTO VOTER_NO-OP;
```

The OCDs here are simplistic because in each case the output characteristic is defined regardless of the input characteristics. Nevertheless, this example illustrates how the OCDs for different failure modes can be combined.

The OCDs defined for the voter under a non-failed state were given in section 3.1, [12], and is repeated here:

```
OUTPUT y:n IF ALL(x(i):n)
OUTPUT y:g IF #(x(i):g) > #(x(i):b)
OUTPUT y:b IF #(x(i):b) > #(x(i):g)
```

The combined OCD for y:g, y:b, and y:n is:

```
OUTPUT y:g IF #(x(i):g) > #(x(i):b) AND VOTER_NOF
OUTPUT y:b IF (#(x(i):b) > #(x(i):g) AND
VOTER_NOF) OR ({} AND VOTER_BAD)
OUTPUT y:n IF (ALL(x(i):n) AND VOTER_NOF)
           OR ({} AND VOTER_NO-OP)
```

where the {} indicates that no input conditions existed for that failure mode's OCD. Eliminating the {}, the new OCDs are:

```
OUTPUT y:g IF #(x(i):g) > #(x(i):b) AND VOTER_NOF
OUTPUT y:b IF (#(x(i):b) > #(x(i):g) AND
VOTER_NOF) OR VOTER_BAD
OUTPUT y:n IF (ALL(x(i):n) AND VOTER_NOF)
           OR VOTER_NO-OP
```

All OCDs may be combined in this way to produce a

correct, combined set of OCDs for the component. This, along with the transitions for each failure mode constitutes the local reliability model that is returned.

In order to reduce the OCDs, heuristics may be applied. Theses heuristics are based on semantic knowledge of the condition variables (e.g., conditions X_BAD and X_NO-OP refer to the same component failure mode and conditions x:b, x:g, and x:n refer to the same variable characteristics). With this knowledge OCD clauses may be combined and possible conditions eliminated so that the resulting OCD is reduced.

Little work has been done at present on reducing the OCD via these heuristics. However, one such heuristic is defined.

This heuristic groups OCD clauses (in disjunctive normal form) according to common input characteristics (i.e., clauses that differ only by component state). If the group of clauses represents all states for that component, then the group may be replaced by a single clause with the input characteristics only. Then, to maintain consistency, a condition, AND NOT({input conditions}) is added to other clauses. An example of this heuristic follows:

Given the OCD:

```
OUTPUT N IF ((r=0 OR r=1) AND XNO-OP)
        OR ((r=1) AND XBAD)
        OR (r=1 and XNOF)
```

change the OCD into disjunctive normal form:

r=0 AND XNO-OP [1]

r=1 AND XNO-OP [2]

r=1 AND XBAD [3]

r=1 AND XNOF [4]

where [1], [2], [3], and [4] identify the new clauses. Attempt to group the clauses according to a common input characteristic:

r=0: Only clause 1 contains this characteristic, and all states of component X are not referenced.

r=1: Clauses [2], [3], and [4] can be grouped:

r=1 and (XNOF,XBAD, XNO-OP)

Since all component states are represented in the grouped clause, the group may be replaced by:

r=1

Next, the condition:

NOT(r=1)

which is equivalent to r=0 may be eliminated from the other clause ([1]) so that the resulting OCD is:

OUTPUT N IF XNO-OP [1]

OR r=1 [2-4]

4.5. MODEL REDUCER/ENCODER

The Model Reducer/Encoder converts the top-level reliability model created by the Model Builder into an ASSIST input file. The reliability model produced by the Model Builder identifies all significant transitions as a result of failure modes and system reaction to failures (i.e., FDIR). Further, it defines the system states that represent the undesirable condition analyzed. The encoding of this model into ASSIST syntax may be accomplished in a relatively straightforward manner. However, ASSIST allows a rich set of primitives in which several representations for the same model are possible. The Model Reducer/Encoder applies heuristics to optimize the ASSIST model, resulting in a reduced SURE state space. For example, instead of modeling separate transitions for each redundant component in the system, the state vector includes one array element representing the number of redundant components in each state. Then, only one transition for each failure type need be encoded for redundant components as a group. Further, it looks for variables that might have been included in the state space but are not required for a correct model.

The separation of the Model Reducer/Encoder from the Model Builder also allows the flexibility to interface with other Markov-based reliability evaluation tools.

5. ENHANCEMENTS

This report describes a general mechanism by which reliability models are generated from a functional and structural specification of a candidate architecture design. Further enhancements to this framework are conceivable.

First, an extension to the BBD/SYSD and their interfaces could allow assertions on the interrelationships between components that are not directly defined as a functional dependency. Consider a system in which devices communicate through a network of communication nodes interconnected by redundant, spare links. Upon a network node failure detection, a network repair process is invoked to enable spare links between the nodes so that no device attached to the network is isolated. Instead of modeling a specific implementation of the repair process, an "abstract" model could be defined at an appropriate level in the BBD hierarchy, whereby the effects of the repair are defined on all subcomponents at that level. For example, the repair abstract model could state that after repair, only devices directly connected to the failed link or node are isolated. This assumes that there are enough spare links in the network that network repair will be able to successfully reconfigure all intermediate node and link failures. The "abstract" model differs from a component model defined in the BBD in that the abstract model specifies changes to many components, whereas a component model BBD is localized for that component. Using abstract models allows a user to

specify changes to the system without defining the implementation of components that bring about the change. This is useful at early stages of the design when implementation details are not well defined, and at later stages when a more compact model is analyzed. The disadvantage of abstract models is that they constitute assumptions on the interaction of components that may not be supported by the implementation. Their use, therefore, should be noted (with all other assumptions) in the final model.

Second, as previously stated, reliability analysis should be performed at all levels of a design. Typically, several levels of functional requirements exist above the highest level at which hardware components are defined. These levels describe what requirements exist (e.g., engine control, flight control, propulsion, etc.). Each requirement is decomposed into subrequirements, until at some point in the design, implementation decisions are made that map the requirements into hardware components and a configuration description. At present, the designer analyzes this level of the hierarchy manually to determine the corresponding component functional requirements. An automated approach incorporates into the SYSD a hierarchy of system level requirements above the component configuration. The format for these requirements could be expressed as logical conditions, or assertion, on the required operations (for example, at least two engines are required for safe flight). Each requirement would be decomposed into subrequirements

until an implementation is defined to meet the requirement. The Reliability Model Generator could then trace a high level requirement to the component functions that implement the requirement and generate the corresponding reliability model for a top level functional requirement. Figure 8 illustrates this additional level of the SYSD.

Third, allow the user to specify local reliability models for some components, and then use the Local Model Generator to generate local reliability models for components for which no local reliability model has yet been defined. (The present algorithms assume the the Local Model Generator defined local reliability models for all components at the lowest level).

Finally, the development of the Reliability Model Generator is part of an ongoing effort to support the design and analysis of highly fault-tolerant control system architectures suitable for high-performance aircraft of the 1990s and beyond. Figure 9 illustrates a long-term plan to incorporate analysis tools into the design and evaluation of highly fault-tolerant systems. The Reliability Model Generator is an integral part of that plan. Another similar tool is envisioned to map a block diagram for a system into the performance models required for simulation studies of a candidate architecture. A third tool is needed to aid in the synthesis of a configuration, given a building blocks architecture and a set of requirements for the system. The three tools should be integrated into a uniform environment

with common interfaces among them.

APPENDIX A: Changes to algorithms since 9/87

The algorithms for both RMAS and the LMG are listed in appendix B. However, because the algorithms have changed since they were presented to NASA Langley on September 10, 1987, appendix A lists the changes that were made to the algorithms. Appendices C and D include traces of the algorithms for two example configurations.

The following is a list of changes that have been made to the algorithms since they were presented to NASA Langley on September 10, 1987.

A.1 Separation of RMAS and LMG

As discussed in section 3.1.2.4, earlier designs of the Reliability Model Generator did not distinguish the Local Model Generator from the Reliability Model Aggregation System. Rather the global model was generated coincident to the failure modes effects analysis (FMEA). Refer to that section for a discussion of the motivation behind that change. Reference [1] for a high-level description of the process flow in the combined approach. Because of this separation the new algorithms have been separated as follows:

OLD PROCESS	NEW PROCESS
Process 1	Process I1 in LMG
Process 2	Combined with Process I1 in LMG
Process 3	RMAS process (includes elements of old Process 1 and 2 to organize trace)
Process 4	MODEL REDUCER/ENCODER
Process 1x	Deleted, see next section
Process 2x	Process L1 in LMG
Process 3x	Process L2 in LMG

Also, the process names are now prefixed with an "I" or an "L" to indicate intermediate or lowest level processes.

A.2 Deletion of Process 1x

The failure mode specification was defined as a change to the function that occurred when the component failed. The purpose of process 1x was to change the function definition for the component according to a failure mode specification. However, it has since been decided that the user can specify the failure mode as a new function thus eliminating the need for process 1x to define the new function. Since most failure modes are defined by their effect on the outputs of the component, this specification is straightforward.

A.3 Changes to RMAS module (previously Process 3)

1. Because the RMAS module has been separated from the other processes, additional statements were added to the module to maintain the hierarchical flow of the RMAS module. This was previously performed by processes 1 and 2. In order to accomplish this, when the module is invoked for an intermediate level component (parent), the subcomponent that generates the parent's output is defined from the BBD for the parent. Then an RMAS module is invoked recursively for that subcomponent which returns a reliability model of its structure (which may include analysis of subcomponents also). Consequently, statement A0 of the RMAS module has been added to invoke the hierarchical structure of the RMAS modules.

2. Besides recursive invocation of RMAS modules for intermediate level components, the RMAS processing for intermediate level models has been changed as follows. The transitions returned for intermediate level models may contain conditions that are not referenced in the OCDs for the models. Thus, in order to trace all effects of a given output characteristic for the intermediate level component, the conditions for the transitions must be analyzed also. This is explained in detail in section 3.3.2.7. Prior to this change, an intermediate level model was simply returned by task 2 of the RMAS. This change does not affect the validity of the traces in appendices B and C since the transitions returned from the intermediate models did not contain conditions that were not referenced in the OCDs for the intermediate level models.

3. The naming of data structures TO_BE_ANALYZED, BEING_ANALYZED and HAS_BEEN_ANALYZED that were used to maintain an order of analysis of conditions has been changed to FUTURE_CONDITIONS, PRESENT_CONDITIONS, and PAST_CONDITIONS lists (or FUTURE, PRESENT, and PAST for short).

4. The names of the tasks within this module have been changed. What was called %:ANALYZE_COMPONENT is now known simply as Task 1, and what was called @:ANALYZED_CONDITION is now known as Task 2. The symbols % and @ were used as a shorthand reference to these tasks in the detailed traces of appendices C and D. Also, the algorithm statements were numbered for reference in the traces. These symbols will remain in the new algorithms in appendix B along with the same numbering convention.

5. The OCD model combination rules for option 2 (see section 3.3.2.3) were changed to nested transitions. The

motivation for this is as follows. Recall the example given in section 3.3.2.3, figure 14. The old rules for option 2 would have created a combined OCD and transition:

```
AGGREGATE OCD for parent of A and B:

OUTPUT y:b IF x:b and q:b and A_FAIL
IF B_NOF and w:b TRANTO B_FAIL, x:b
```

Here, the condition part of the transition is appended with the extra condition "w:b" and the consequent part of the transition is appended with "x:b". This is incorrect in the case where B_NOF is true and w:b is not, since the transition to state B_FAIL, that should occur, would not. The correct transition is one in which the condition B_NOF alone determines if the transition to state B_FAIL is possible, and then the added condition w:b determines if a transition to x:b is possible also. This is reflected in the transition change presented for option 2 in section 3.3.2.3:

```
AGGREGATE OCD for parent of A and B:

OUTPUT y:b IF x:b and q:b and A_FAIL
IF B_NOF TRANTO B_FAIL

     IF w:b TRANTO x:b
ENDIF;
```

A.4 Changes to Process L1 (previously Process 2x)

1. The delineation of three phases for the process was not defined in the previous algorithms. The phases were introduced as a natural way to explain the parts of Process L1.

2. Procedural Verses Non-procedural Functions: In the previous version of the algorithm, a clear distinction was made between procedural and non-procedural function specifications, which were defined as Type A and Type B functions respectively. Section 3.1.2.2 discussed the distinction between these two function types. Upon reexamination, it became apparent that the algorithm performed similarly for Type B functions as for Type A functions where the element to the left of the "|" was a variable. That is, in both cases, each possible characteristic for the output variable is instantiated, and a new set of clauses is defined. The only real distinction between the two cases, then, is the presence of quantifiers for non-procedural specifications. Therefore, the code for the two cases was combined so that there is not a separate processing for each. Rather, when a variable is present on the left side of

the "|" sign, the processing is the same except for the handling of quantifiers, if any exist. Refer to section 3.4.3.2.2 for a complete description of phase 2.

3. Propagation Rule Changes: The condition propagation rules that were performed in the previous algorithm were not sufficient to cover all cases. These propagation rules were based on the assumptions that:

   1. A variable whose characteristic is "g" is equal to another variable whose characteristic is "g",

   2. A variable whose characteristic is "b" is equal to another variable whose characteristic is "b", and

   3. A variable whose characteristic is "g" is not equal to another variable whose characteristic is "b".

From these assumptions, the propagation rules were defined to automatically eliminate combinations of variable characteristics which would violate these assumptions. For example, the rule:

Condition:
    var1:(SET1)=var2:(SET2)

Action:
    IF SET1 is 0 THEN var1:(SET2)=var2:(SET2)
    IF SET2 is 0 THEN var1:(SET1)=var2:(SET1)
    ELSE var1(SET1 n SET2)=var2(SET1 n SET2)

eliminated from the characteristic set all combinations that violated the assumptions above.

Although these assumptions were applicable for the traces given in appendices B and C and are applicable in most all cases, there may be situations in which one or more of these may not hold. Therefore, the propagation of characteristics through conditions such as =, <>, etc. is currently performed interactively with the user, as discussed in section 3.4.3.2.1 and section 3.4.3.2.3. Further, the three assumptions listed above, which will apply in most cases, are the first to be verified by the program.

This change makes the Reliability Model Generator a more comprehensive analysis tool.

4. Overlaps: The handling of overlaps within and between characteristics was solidified into a procedural definition that was discussed in section 3.4.3.2.3. Prior to this, the general procedure was introduced and illustrated for a example, but the exact algorithm was not defined. (The traces that were performed in appendices B and C did not result in overlapping

conditions.) Further, because of the need to verify assumptions (previous item) with the user interaction with the user was added in the algorithms for resolving conflicts.

A.5 Changes to Process I2 (previously Processes 1 and 2)

1. As stated in item 1, processes 1 and 2 were combined into a single process I1.

APPENDIX B: ALGORITHMS

The following is a listing of the pseudocode for the algorithms. It is organized as follows:

A. USER INTERFACE: This describes the features of the proposed interface.

    BBD INTERFACE

    SYSD INTERFACE

B. RELIABILITY MODEL GENERATOR

    1. RELIABILITY MODEL AGGREGATION SYSTEM: This process is identified here and elaborated on at the end of the appendix.

    2. LOCAL MODEL GENERATOR

        PROCESS I1
        PROCESS L1: This process is identified here and elaborated on at the end of the appendix.
        PROCESS L2

C. MODEL REDUCER/ENCODER

RELIABILITY MODEL AGGREGATION SYSTEM: algorithm listed separately

PROCESS L1: algorithm listed separately

DATA TYPES

A. USER INTERFACE

BBD INTERFACE

INPUT: Graphical, interactive

OUTPUT: (1) Building Block Definition - BBD

PROCESS: Convert user friendly templates to BBD; interactive input, graphical

1. Automatically create subcomponents' BBD spec shell from subcomponent interaction field of parent

2. BBD editor has compiler that checks: input/output function consistency, etc. , primitive spec(no missing ;)

3. alter at will:

4. generic failure modes such as no-op and bad-data that can be created for a function by just specifying the name, and the editor fills in the details

5. Availability of a set of macros and operation for user friendly functional specification. Availability of a mechanism to create and edit macros.

See also Reliability Model Generator User's Guide, June 15, 1988.

A. USER INTERFACE (cont.)

SYSD INTERFACE

INPUT: Graphical, Interactive

OUTPUT: (2) System Definition - SYSD

PROCESS:

1. Identifies individual components of the configuration by:

   - individual name for each component
   - reference component type (name of component in BBD)

2. Identifies interdependencies of components (connectivity) if not already specified in BBD as standard connection

3. System fills in what it know of component from BBD and queries for additional information needed:

   - # of each redundant sub
   - other connections it needs

4. Determines lowest level and highest level of functional abstraction to be analyzed.

5. System looks for faults at lowest level - if none, queries

B. ANALYZE PROCESSES:

RELIABILITY MODEL AGGREGATION SYSTEM:

INPUTS:

(1) BBD
(2) SYSD
(3) trace tables of subcomponents
(5) Local Reliability Models for Lowest Level Components

OUTPUT:

(5) Top-level reliability model for system

PROCESS: defined separately at end of appendix

B. ANALYSIS PROCESSES (cont.):

LOCAL MODEL GENERATOR

INITIALIZER:

GIVEN: Unreliable condition references component output: already defined high level component to instantiate

PROCESS:
IF LOWEST LEVEL REACHED (look at configuration definition - SYSD)

INSTANTIATE PROCESS L1,

ELSE INSTANTIATE PROCESS I1

B. ANALYSIS PROCESSES (cont.):

LOCAL MODEL GENERATOR (cont.):

PROCESS I1:

1. FMEA - TRACE SETUP:

INPUT:
(1) BBD
(3) Function to trace given by parent
(3) Input Characteristics

INTERNAL DATA STRUCTURES:
(4) Priority Queue
(3) subfunction trace tables

PROCESS:

Based on functional Flow, identify order of subfunctions to trace. Maintain order in priority queue (6). For each subfunction, identify outside effects relevant to subfunction

IF function to trace has not been analyzed OR
IF input characteristics have not been analyzed

1. sets up subcomponent trace tables:
   a. from BBD, identify each subcomponent involved in given function
   b. input characteristics for each subcomponent

2. set up priority queue by referencing subfunction and determining order of subcomponents to trace

   DETERMINE ORDER:

   R1. IF a;b order(a,b) where a and b are component

   R2. For cyclic subprocesses: (determined by 'cyclic' in function)

      1. Find ending point (subcomponent whose effects are seen outside

      2. Find starting point (subcomponent whose inputs come from outside)

      3. Must loop trace until no new effects are found (or maximum number of failures considered)

   R3. For fully redundant subcomponents,

analyze only once (full redundancy requires identical inputs and trace histories.

3. Loop until (output of parent reached and no subcomponents on priority queue)

   A. TAKE TOP ELEMENT OFF QUEUE Pass input characteristics to subcomponent process

   B. Upon return of subcomponent module:

      Update input characteristics of trace tables for affected subcomponents based on OCDs of subcomponent module just returned

      - for parallel functions, give common characteristics
        - ex. FA x(i) of P(n) char = {...}

      - Heuristics to Ignore Irrelevant Effects
        - based on likelihood
        - based on number of failure modes in history
        - based on redundant information

B. ANALYSIS PROCESSES (cont.):

LOCAL MODEL GENERATOR (cont.):

PROCESS L1:

ANALYZE ALL COMBINATIONS OF FAULTS AND OUTSIDE EFFECTS ON FUNCTION

INPUTS: (3) input characteristics

OUTPUTS: (5) Local reliability model for this component

PROCESS: Defined separately at end of appendix

B. ANALYSIS PROCESSES (cont.):

LOCAL MODEL GENERATOR (cont.):

PROCESS L2:

DEFINE LOCAL RELIABILITY MODEL: called after process L1 has been performed for all fault behaviors

1. If there are no input characteristics considered, add no-fault state:

   STATE GOOD IF x_NOF {where x is component name}

2. Combine conditions for every unique output characteristic

   STATE (output characteristic)
   IF (OCD in L1) AND (fault state)
   OR {next fault state}

3. REDUCTIONS:

   1. Reduce STATE description where () is null

   2. The following are heuristic that can reduce the number of fault conditions considered when an output characteristic is primarily input characteristic driven:

      1. put in Disjunctive Normal Form

      2. group clauses according to common input characteristics (* in other words clauses that differ only by component state *)

      3. for each group: if all state possibilities represented in clauses:

         - replace group by single clause with input characteristics only

         - eliminate ^(input characteristic) reference in other clause

      Example:

      OUTPUT N IF ((r=0 OR r=1) AND XNO-OP) OR ((r=1) AND XBAD)
      OR (r=1 and XNOF)

      1. r=0 AND XNO-OP [1]
         r=1 AND XNO-OP [2]

```
        r=1 AND XBAD     [3]
        r=1 AND XNOF     [4]

    2.  for r=0: not fully redundant

    2.  for r=1:  r=1 and (XNOF,XBAD,  XNO-
        OP) [2-4]
        - fully enumerated component state
        - reduce  [2-4] to r=1
        - eliminate r=0 from [1]

    RESULT:
        r=1 OR XNO-OP

4.  return to parent
```

C. MODEL REDUCER/ENCODER

INPUTS:

(7) Reliability Model from Top level component

OUTPUTS:

(8) RELIABILITY MODEL (ENCODED FOR RELIABILITY TOOL)

PROCESS:

This process uses techniques to reduce the number of states generated The number of states generated by the model is dependent on the # of transitions and the number of possible states from which the transitions may occur. Both the Reliability Model created by the foregoing processes and the reduced model created here should be accessible to the analyst, because with model reduction comes a loss of information

1. Truncation: truncate the model after the xth failure level

   - can be added to model as separate state vector element and additional condition of death state

2. Pruning: do not consider faults or failure sequences that are unlikely.

3. Heuristics: these are the 'tricks' that has been applied in manual model generation. The model inputed to this process contains semantic information about the components that may be exploited to reduce the model. Examples of semantics:

   1. P_BAD, P_NO-OP : identifies these two variables as relating two the same component P and also relating to a fault state of the component P. Heuristics may under some situations combine these variables into one variable for component P

   2. y:(g), y:(b) : identifies an output characteristic for y Heuristics may under some situations combine these variables into one variable for y. They may also be able to combine with the component that depends on the variable.

Examples of Heuristic application:

1. IDENTIFY SUPERFLUOUS VARIABLES:

a. FOR EACH change in changes:

IF the change is not referenced in the condition part of other transitions

AND NOT in the overall death state definition for the component

THEN mark those variables as superfluous (i.e., not needed)

b. COMBINE superfluous variables (with same common prefixes and suffixes of the component referenced) into one variable characteristic (called ANYOF)

c. Change all references to the superfluous variables to ANYOF

Ex: In example2 PiBAD and PiNO-OP are superfluous so P_ANYOF(Bad,Nil) is substituted in

d. IF the only state for a component or data characteristic for a variable is ANYOF,
THEN its reference may be eliminated (at the expense of possible loss of information in the resulting model)

2. FOR REDUNDANT COMPONENTS:
IF Xi_NOF is only cited as a precondition for any transition
AND all other states of X are superfluous,
THEN

a. create a variable counter (#XNOF) and substitute #XNOF for all occurrences of IF x(i)_NOF,

b. add X #XNOF to rate

c. substitute #XNOF=#XNOF-1 in for superfluous variables in that transition. Get rid of other superfluous variables

ENCODING RELIABILITY MODEL INTO RELIABILITY ANALYSIS TOOL:

EXAMPLE: ASSIST

STATE VECTOR ELEMENTS:

1. The state vector element for each component state is created. The number of values possible for the element will equal the number of states for the component with

value 0 being the NOF state. For redundant components the SYSD will determine the number of individual components to be represented.

Ex. B[0..1] (* 0=BNOF, 1=BNO-OP *)

2. For operations, the model for the operation determines the way it will be represented. For example, the model for the operation, #:

MODEL: ALL(<cond>)

COI: <cond>

Change rep: #(<cond>)=?get max(obj) from BBD where obj is component (or component of data) in <cond>

Contributory transitions:

replace IF (<cond>)

with IF #(<cond>) = Max(obj)

replace TRANTO <cond>

with TRANTO #(<cond>)=#(<cond>) + 1

Detractory transitions:

replace <cond>

with #(<cond>)=#(<cond>) - 1

ASSIST CONVERSION: #(<cond>) : [0..max(obj)]

3. For variable characteristics, create a state vector element similar to the vector elements for component states.

DATA TYPES:

1. Building Block Definition (BBD): The BBD is the internal representation which contains a hierarchical representation of the components so that the functional flow may be traced and various levels of detail may be analyzed for different failure conditions.

FUNCTIONAL SPECIFICATION: the functional flow of the component which identifies its inputs, outputs and the relationship between them.

STRUCTURAL REQUIREMENTS: Specifies what configurations of BBD components are allowed and what configurations aren't allowed.

NEXT LEVEL VIEW: For high level components, this identifies the subcomponents involved in the functional specification, and their interaction relationship.

FAILURE MODE SPECIFICATION: This specifies the manner in which a component may fail in terms of its effects on the data that comprise the function. Any data value may be set to the characteristics b (bad or corrupted in some manner) or n (non-existent or not available on time) For example, the following faults may be modeled:

	b	n
data:	corrupting input port	disabled input port input not available on time
internal data:	bad function	disabled function
output data:	corrupted component corrupted port	disabled component disabled port

2. System Definition (SYSD): Defines configuration of BBD components for a candidate configuration.

Configuration Definition:

a. BBD instantiations: user can specify:

   1. Components BBD name: BBD type reference

   2. Lower level to specify for named component

      - System fills in what it know of next level and queries for additional information needed

      - # of each redundant sub

      - Other connections it needs

3. Interconnections between specified components

3. Trace Table: one for each functional process at each level of abstraction

Function	Input Char.

Function: particular function involved in trace
Input characteristics: any relevant effects of other components on this function

4. Priority Queue: Contains an ordering of subfunctions to be traced. Primarily ordered by functional flow; but contains arbitration ordering for parallel subfunctions, or possibly other heuristic to limit tracing complexity.

subcomponent	priority

5. Local Reliability Model: One for each functional process at each level of abstraction; For the highest level component, this is the primary analyzer (ASSIST).

```
OUTPUT DEFINITION
     OUTPUT <output var>:<char> IF <cond>
TRANSITIONS:
     IF <cond> TRANTO <cond> BY <rate>
          ...                BY T<cycle>
<char> :: g | b | n
<cond> :: <op>(<cond>) | <cond> OR <cond> | <cond>  AND
<cond>
          | <input var>:<char> | <fault>
<input var>,<output var>  :: <var>
<var> :: lower case
<fault> :: <compname>-<var>
<compname> :: upper case
<op> :: # | ALL | ... for all defined operations
<rate> :: <fault>_RT
<cycle> :: {<compname>}+   (* one or more component
names *)
```

(6). Reliability Analysis Tool syntax : (ASSIST)

```
;; NOTE:  ++ n ++ (where n is a number from 1 to 7) indicates
;;            a reference to an explanation of the reasoning involved
;;            in this part of the algorithm.  The explanations are in
;;            a file called RMAS.discussion.

RMAS (COMP)

;; where COMP is the parent component

TOP: FOR EACH OF COMP'S OUTPUTS
     1.  find the subcomponent that outputed data (SUB)
     FOR EACH SUBCOMPONENT OUTPUT CHARACTERISTIC DEFINITION (OCD):
     1.  put condition on BASE list
     2.  establish COMP's initial OCD:
           OUTPUT <CHAR> IF <outputing subcomponent condition>
     3.  call %:ANALYZE_COMPONENT (COMP SUB CHAR)

%:ANALYZE_COMPONENT (COMP SUB CHAR)  - TASK 1
;;DEFINITIONS:
;;  CHAR references output characteristic for component SUB
;;  where COMP is parent of SUB
;;  DEFINE:
;;    Output Characteristic Definition (OCD):
;;           OUTPUT <CHAR> IF <condition>
;;  PROCESS FLOW:
;;  IF  COMP is intermediate level: call RMAS submodule to define
;;      reliability model for that level.  Then interrogate this module
;;      to find transitions that contribute to input conditions for
;;      component
;;  ELSE
;;      Find the transitions that contribute to output characteristic by
;;      calling @:ANALYZE_CONDITION recursively to find the transitions
;;      that contribute and detract from each condition in the OCD for
;;      CHAR.  This involves a trace the OCDs of all components that
;;      are subcomponents to COMP until the inputs of COMP are reached
;;      or a cycle is detected.  When a cycle is detected, this procedure
;;      determines if the cycle involves a non-fault (recovery) transition
;;
;;  1.  disect OCD condition into disjunctive normal form clauses (DNF)
;;      and determine which conditions in each clause must be analyzed
;;      - see case statements
;;  2.  call @ANALYZE_COND (cond) for each cond to be analyzed
;;  3.  some transitions returned will be altered to account for states
;;      of other clauses (see case statements)
```

```
%:ANALYZE_COMPONENT (COMP SUB CHAR)  - TASK 1
   A. IF SUB is a parent component:
++ 7 ++ 0.  Invoke RMAS module for subcomponent. Returns local reliability
             model UPON RETURN:
        a.  For each CLAUSE in new COND (separated by OR)
            1.  if any cond is on PRESENT_COND list - (ignore clause)
                (trace was a cycle from which no transitions were found)
            2.  if any conditions are on the 'PAST_COND' list:
                 1.  substitute BASE equivelents in for logicals
++ 1 ++          2.  call DNF (CLAUSE) - disjunctive normal form
            3.  apply the appropriate case statements to CLAUSES to define
                a list of conditions to analyze:
                  (analyze clauses involved in substitutions first)
                case 1:  OUTPUT cond IF cond:
                   1. Ignore clause: (condition substituted is equivelant
                      to the condition being analyzed under %)
++ 1 ++            2. any transitions found for other CLAUSES must
                      include IF ... AND ^cond ... TRANTO ...
                      (with the exception as noted in case3, #2)
                case 2:  OUTPUT cond IF cond AND A:
                ;; A is the remaining conditions in the clause
                   1. ignore clause (conditions in A are not analyzed)
                   2. any transitions found for other CLAUSES must
                       include IF ... AND ^cond ... TRANTO ...
                case 3:  OUTPUT cond IF ^cond
++ 5 ++            1. add a non-fault transition:
                       IF ^cond TRANTO cond BY T(CYCLE)
                case 4:  OUTPUT cond IF ^cond AND A
++ 5 ++            1. add a non-fault transition:
                      IF A AND ^cond TRANTO cond BY T(CYCLE)
                   2. call @ANALYZE_COND(cond) for conditions in A
                      satisfying 1 of case 5 (transitions returned NOT
                      subject to stipulation 2 of cases 1 and 2)
                case 5:  OUTPUT cond IF (conditions that excludes cond)
++ 7 ++            1. call @ANALYZE_COND(cond) for input conditions that
                      are not on 'PAST_COND' or 'PRESSENT_COND' list
++ 1 ++            2. apply stipulations that exist from other clauses
                      to transitions returned (case 1 and 2, #2)
++ 7 ++            3. For each  SUB STATE or SUB_IO_COND (not parent I/O)
                      a. for each transition in CTL for condition
                         1. for each INPUT_CONDITION condition found in
                            condition part of transition
                            a. if INPUT_CONDITION already analyzed
                               1. if INPUT_CONDITION is logical
                                  - substitute in BASE equivelant
                                  (MODEL_COMBINATION_1 was used in
                                   analysis of INPUT_CONDITION for OD)
                               2. if INPUT_CONDITION is BASE - do nothing
                                  (MODEL_COMBINATION_2was used in analysis
                                  of INPUT_CONDITION for OD)
                            b. if INPUT_CONDITION not analyzed
                               1.  analyze (call % INPUT_CONDITION)
                               2.  substitute using MODEL_COMBINATION_1
                                   - should always be able to perform this
```

```
%ANALYZE_COMPONENT(COMP SUB CHAR)  - continued
   B. ELSE
++ 1 ++ 0.  call DNF (OCD) to change OCD to distunctive normal form
        a.  For each CLAUSE in new OCD (separated by OR)
            DEFINE: CLAUSE = cond1 OP1 cond2 OP2 cond3 ... OPn condn
                    OPi = One_of (AND, >, <, =)
            1.  if any cond is on PRESENT_COND list - (ignore clause)
                ;; the trace resulted in a cycle from which no
                ;; transitions were found - return
            2.  if any conditions are on the 'PAST_COND' list:
                 1.  substitute BASE equivelents in for logicals
++ 1 ++          2.  call DNF (CLAUSE) to change to distunctive normal
                     form
            3.  apply the appropriate case statements to CLAUSES to define
                a list of conditions to analyze:
                  (analyze clauses involved in substitutions first)
                case 1:  OUTPUT cond IF cond:
                   1.  ignore clause
                       ;; If condition substituted in is equivelant to
                       ;; the condition being analyzed under %, ignore
                       ;; this clause - no transition found
++ 1 ++            2.  any transitions found for other CLAUSES must
                       include IF ... AND ^cond ... TRANTO ...
                       (with the exception as noted in case3, #2)
                case 2:  OUTPUT cond IF cond AND A:
                ;; A is the remaining conditions in the clause
                   1.  ignore clause (conditions in A are not analyzed)
                   2.  any transitions found for other CLAUSES must
                       include IF ... AND ^cond ... TRANTO ...
                case 3:  OUTPUT cond IF ^cond
++ 5 ++            1.  add a non-fault transition:
                       IF ^cond TRANTO cond BY T(CYCLE)
                case 4:  OUTPUT cond IF ^cond AND A
++ 5 ++            1.  add a non-fault transition:
                       IF A AND ^cond TRANTO cond BY T(CYCLE)
                   2.  conditions in A satisfying 1 of case
                       5 are analyzed - call @ANALYZE_COND (cond)
                       -the transitions returned are NOT subject to
                        stipulation 2 of cases 1 and 2
                case 5: OUTPUT cond IF (conditions that excludes cond)
                   1. analyze conditions (@ANALYZE_COND(cond)) that are
                      not on 'PAST_COND' or 'PRESSENT_COND' list
++ 1 ++            2. apply stipulations that exist from other clauses
                      (case 1 and 2, #2)
```

```
%ANALYZE_COMPONENT(COMP SUB CHAR)  - continued

            4. for each condition to be analyzed: establish order of
            analysis and call @ANALYZE_COND (cond):
                - analyze component states first (B in @ANALYZE_COND )
                - then simple input conditions (A and C in @ANALYZE_COND )
                - then predicate input conditions (D in @ANALYZE_COND )
                - put conditions on 'future' list

END (%:ANALYZE_COMPONENT)
```

```
@:ANALYZE_CONDITION (COND COMP): FIND TRANSITIONS THAT
                          CONTRIBUTE TO CONDITION:    - TASK 2
;; SUMMARY:
;;  Find the transitions that contribute to COND.
;;  This is called recursively by %ANALYZE_COMP to trace an output
;;  characteristic thru all subcomponents until the inputs of the parent
;;  are reached or a cycle is detected.
;;   1.  tracing the source of the condition of COMP to find transitions:
;;       Possibilities for each condition and action taken as a result:
;;       A.  Conditions on PRESENT_COND, or PAST_COND lists:  ignored
;;       B.  Condition based on state of COMP:
;;           fault or nofault case
;;           - transitions found for faults, and
;;           - input characteristics of COMP are established as bases
;;       C.  conditions based on inputs:
;; ++ 3 ++   1.  conditions based on some input predicates
;;               - find simple input characteristics on which the
;;                 predicate is based
;;               - call @ANALYZE_CONDITION to find transitions that
;;                 affect these simple conditions
;;               - change transitions found to reflect predicate
;;           2.  conditions based on a simple input characteristic
;;               -call %ANALYZE_COMPONENT for component that outputed
;;                to this component
;;                (input condition becomes output condition in new
;;                 component    )
;; ++ 2 ++       - change the transition returned to reflect the
;;                 variables in the OCD or change the OCD to reflect
;;                 the variables in the transition (see #2)
;; ++ 4 ++   3.  change transition to reflect a detraction to
;;               any previously defined bases
;;           4.  change previously defined transitions to reflect
;;               detraction from newly defined bases
;;   2.  applying rules to convert transitions to reference variables
;;       defined by output definition
;;       - establish set of BASE variables from which all transitions
;;         are defined - assume all variables are bases unless they
;;         can be eliminated by an equivelent relationship among other
;;         varialbles (this is done in MODEL_COMBINATION_1)
;;         - a condition is put on BASE list
;;           1. for each component state for components that can fail
;;           2. for conditions involved in non-fault transition.
;;       - rewrite or add to transitions found any changes in established
;;         bases
;;       - establish logicals for other data characteristics not defined
;;         in terms of bases so that when referenced again, the base
;;         equivelent may be substituted
;;   3.  for each set of transitions found from @:ANALYZE_CONDITION update
;;       the contributory transitions list (CTL) for that to condition
```

```
@:ANALYZE_CONDITION (COND COMP):

++ 6 ++ A. If condition is on PRESENT OR PAST COND lists - ignore
          NOTE: this test was put in %:ANALYZE_COMPONENT instead of here
          so as to avoid a procedural call, but is still included here
          to be consistent with traces

       B. FOR CONDITIONS BASED ON COMPONENT STATE (fault or nofault) :
          1. take condition off 'future', put on 'PAST_COND' list.
          2. add transition to queue of transitions that contribute to
             condition

             Transition:  IF subname_NOF TRANTO subname_fault
          3. add fault to BASES list
          4. add condition of %:ANALYZE_COMPONENT To logical (not
             needed if subcomponent outputs to outside world
             - represented as output definition instead)

       C. FOR CONDITIONS BASED ON INPUTS:
          1. take off 'future', put on 'PRESENT_COND' list
++ 3 ++   2. If condition is based on input predicates:
             (*  condition =  PRED(...) for some PRED = predicate
             (*  for ex:  MAX(x(i)), AVE(x,y), #(<cond>)
             look at model for predicate:
             1. if representation changes:
                a. if condition is a base:  replace base with new
                   representation on base list
                b. add representation to list of conditions to analyze
                c. call @:ANALYZE_CONDITION with new rep
                   1. add transitions found to contributing
                      transition list (CTL) for this condition
                d. move condition (old and new rep) to 'PAST_COND' list
             2. (Rep not changed ) define input condition(s) of
                interest (COI) - add COI to BASES list
                a. add to list of input condition(s) to be analyzed
                b. call @:ANALYZE_CONDITION with each condition
                   1. evaluate transitions and/or output definition
                      according to rules (contained in model for
                      predicate)
                   2. add transitions found to CTL for this condition
                d. move condition to 'PAST_COND' list
             ELSE go to 3.
```

```
@:ANALYZE_CONDITION (COND COMP): - continued

        C.  3.  FOR SIMPLE INPUT CONDITION ( not based on predicate )
                (FIND transitions that contribute to input characteristic)
                1. Determine which component (OUTPUT_COMP) outputed this
                   condition to the current component (CURRENT_COMP)
                   - (look at BBD/SYSD):
                   DEFINE CURRENT_COMP and OUTPUT_COMP as referenced
                   a. if OUTPUT_COMP is PARENT (defined in TOP) - return
                   b. if OUTPUT_COMP is another subcomponent
                      - call %:ANALYZE_COMPONENT(OUTPUT_COMP cond)for
                        component that outputed data
                        ;; input condition becomes output condition in new
                        ;; component
                2. matching output definitions and transitions:
                   a. for non-fault transitions returned - no changes
++ 2 ++            b. if cond not on BASE list (not COI for a predicate)
                      or not input to faulted component:
                         call MODEL_COMBINATION_1
                         - if not successful - call MODEL_COMBINATION_2
                   c. IF COI for an predicate
                      - call MODEL_COMBINATION_2
                3. move condition to 'PAST_COND' list
++ 4 ++     4. if condition is a BASE, perform #FIND_DETRATIONARY_EFFECTS

END (@:ANALYZE_CONDITION)
```

```
MODEL_COMBINATION_1(CURRENT_COMP, OUTPUT_COMP condition)
++ 2 ++
  1.  define  condition (% cond just called)
      as logical according to output characteristic definition returned
      from OUTPUT_COMP and take this condition off BASE list
  2.  replace condition in parent's output characteristic definition (OCD)
      with condition in OUTPUT_COMP's output definition
  3.  No change to OUTPUT_COMP's transitions


MODEL_COMBINATION_2 (CURRENT_COMP, OUTPUT_COMP condition)
++ 2 ++
    ;; USE RULES TO TRANSLATE TRANSITIONS FOUND INTO TRANSITIONS THAT
    ;; REFERENCE OUTPUT DEFINITION:
         Current comp: OUTPUT ... IF ... AND/OR P ...
         want to keep reference to P, so when OD for
         P returns from another comp (OUTPUT_COMP) that outputs P,
         change transitions returned  from OUTPUT_COMP
         to transitions that
         explicitely tranto P

      OPTION 1.  GIVEN:  OUTPUT P IF A AND B
         for all transitions found: IF q TRANTO A and r
            substitute IF q TRANTO A and r
                          IF B TRANTO P
                       ENDIF
         for all transitions found: IF q TRANTO ^A and r
            substitute IF q TRANTO ^A and r
                          IF B TRANTO ^P
                       ENDIF

      OPTION 2.  GIVEN:  OUTPUT P IF A OR B
         (* Note here that in order to define a transition to P,
            P must not already be true, and therefore, a transition
            to A causes a transiton to P ONLY IF B is not already
            true - thus, the logical OR in output definitions must
            be changed to logical XOR in transitions. *)

         for all transitions found: IF q TRANTO A and r
            substitute IF q TRANTO A and r
                          IF ^B TRANTO P
                       ENDIF
         for all transitions found: IF q TRANTO ^A and r
            substitute IF q TRANTO ^A and r
                          IF ^B TRANTO ^P
                       ENDIF
```

```
++ 1 ++ DNF(cond)

;;  This procedure translates a comples boolean formula into distunct
;;  normal form (or SUM OF PRODUCTS).
;;  This functions redefines each boolean expression in order of its
;;  precedence in the boolean formula and applies the following rules.
;;  Assume that the AND has precedence over OR in the absence of ().
;;  Here the | indicates the OR logical function which is used to
;;  illustrate ;;  the separate clauses that are returned.

      A AND B  --> no change                        | A AND B

      A OR B   --> no change                        | A
                                                    | B

      (A OR B) AND C --> A AND C                    | A AND C
                         OR B AND C                 | B AND C
```

```
++ 4 ++ #-FIND-DETRACTORY-EFFECTS:

;; Called at end of @:ANALYZE_CONDITION in F to update transitions to
;; reflect detractions from conditions that are input predicates:
;;      1.  change transition to reflect a detraction to
;;          any previously defined bases
;;      2.  change previously defined transitions to reflect
;;          detraction from newly defined bases
;; (for example, suppose a transition was found that contributed to the
;; number of bad inputs being increased.  #B=#B+1;  since there is a base
;; #G, that number would be decreased.
;;

(#) FOR EACH NEW TRANSITION IF ... TRANTO ...:
    ;; Are there any previously defined bases that are affected by this
    ;; transition?
    A. FOR ALL (previously defined) OPERATIONAL BASES (that represent
       input predicates)

          1.  If COI, look at logical for COI

          2.  if base or logical defined in IF ... part of transition
               (as a high level AND) and

              if base or logical violated in (TRANTO...) part of transition

              THEN ADD TO THE (TRANTO ...) USING  THE DETRACTORY RULES FOR
              THIS CONDITION

       B.  FOR EACH NEW OPERATIONAL BASE ESTABLISHED (* are there any
           previously defined transitions that affect the base *)

           FOR ALL (previously defined) transitions

           1.  If COI, look at logical for COI
           2.  if base or logical defined in IF ... part of transition
                (as a high level AND) and
               if base or logical violated in (TRANTO...) part of transition
               THEN ADD TO THE (TRANTO ...) USING  THE DETRACTORY RULES FOR
                  THIS CONDITION


                        PROCESS L1.alg:


;; PROBLEM DEFINITION:
;; 1.  Given component function (already modified for fault behavior and
;;     defined in terms of input values) and
;; 2.  given input characteristics
;;
;; Determine:
;;   - identify output characteristics given input characteristics and
;;     fault state
```

```
;;    - eliminate references to functions local variables and references to
;;      output variables to right of |
;;    - eliminate quantifiers
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
```

```
PROCESS L1.alg: (cont.)

FOR EACH SENTENCE IN FUNCTION DEFINITION: (* delineated by ; *)

(* follow propagation rules to propagation input characteristics
(* to output characteristics.
(* GOAL: end up without quantifiers, without variables except inputs

PHASE 1:

 1.  FOR EACH FUNCTION DOMAIN: delineated by |

        1.  substitute characteristics in for variables:
            a. for each characteristic substituted in, add
               the conditions of other variables (from other sentences
               under which the substitution can be made
            - order of substitutions
              1.  quantifiers first- after quantifier has
                  been propagated, substitute into action part
                  of function and propagate again recursively
              2.  data that is dependent on others (from previous
                  sentences)
                  ex:  x:(g) | y:(b)  :  substituting in
                     for x:(g), must include y:(b)
              3.  others

        2.  apply propagation rules to operations or
            conditions affected by substitutions
            (this includes propagation of other characteristics
            included in the condition of the substitution)
            a. if object is function - apply function propagation
               rules for characteristics (only when all parameters in
               function have characteristics):
               a.  if function not identified as nil sensitive or not
                   nil sensitive (NS or NNS),
                   use a defined DEFAULT to translate to NS or NNS
            c. if object = condition, use condition propagation rules

        ;;  after each substitution/propagation, substitute resulting
        ;;  characteristics in for other occurrences of variable, and
        ;;  propagate through these occurrences until entire effect of
        ;;  substitution has been propagated

        ;;  redundant data: if propagation results in characteristic
        ;;  being assigned to variable that is redundant, except
        ;;  in the case of a quantifier (FA), do not substitute
        ;;  characteristic in for other occurrences of redundant
        ;;  variable since characteristics between instantiations of
        ;;  redundant variables may be different:
        ;;  ex: x(1):(b) and x(2):(g)

        ;;  this procedure will be implemented in a simpler
        ;;  manner with the object oriented structure of KEE
```

PHASE 1 RULES:

```
*************************************************************
* SUBBING RULES:  for substituting input characteristics    *
* in for input variables                                    *
* SUBSTITUTIONS: VAR is variable being substituted, SET     *
* contains the characteristics:                             *
* 1.  VAR:(set) --> VAR E SET2 ==>                          *
*   if set2=0 --> var:(set)                                 *
*         else var:(set n set2)                             *
*             if (set n set2) = 0 --> USE SET DEFINED       *
*                   BY FAULT/FUNCTION                       *
*************************************************************
*****************************************************************
*  NIL SENSITIVE FUNCTION PROPOGATION RULES:  for GOOD,         *
*  BAD, NIL characteristics for y = O(x1,x2,...xn)              *
*                                                               *
*  CONDITION:                          ACTION:                  *
*  1.  ALL x:(g)          y:(g)  | AND(x1:(g), x2:(g) ...       *
*                                      xn:(g) )                 *
*  2.  TE x:(n)           y:(n)  | OR(xi:(n)                    *
*                                    (AND xj:(^n) FA j<i)       *
*                                    (AND xk:(same) FA k>i)     *
*                                  FA xi:(n) )                  *
*  3.  ^TE xj:(=N),       y:(b)  | OR(xi:(b)                    *
*      TE x:(b)                      (AND                       *
*                                      xj:(set-n) FA j>i )      *
*                                    (AND xk:(g) FA k<i)        *
*                                  )                            *
*                                FA xi:(b),                     *
*                                FA k<i:xk:(set - g <> set)     *
*****************************************************************

*****************************************************************
*  NON-NIL SENSITIVE FUNCTION PROPOGATION RULES: for GOOD,      *
*     BAD, NIL characteristics for y = O(x1,...,xn)             *
*  where O is not nil value sensitive:                          *
*                                                               *
*  CONDITION:                          ACTION:                  *
*  1.  ALL x:(g)          y:(g)  | AND(x1:(g), x2:(g) ...       *
*                                      xn:(g) )                 *
*  2.  TE x:(b)           y:(b)  | OR (xi:(b)                   *
*                                    AND (xk:(^b) FA k<i)       *
*                                FA xi:(b),FA k<i:xk:(g)        *
*  3.  ALL x:(n)          y:(n)  | AND(x1:(n), x2:(n) ...       *
*                                      xn:(n)                   *
*****************************************************************
```

PHASE 1 RULES: (cont.)

```
*  CONDITION PROPAGATION RULES:  for conditions                    *
*  =, <> , < , > , >= , <= :  <rel>                                *
*  Assumptions:                                                    *
*  1.  assume x:g <> y:b  where x,y are any variables              *
*      - for x:(set1) = y:(set2)                                   *
*      - separate into four possible clauses:                      *
*      1.  if g in set1 then remove b from set2                    *
*      2.  if b in set1 then remove g from set2                    *
*      3.  if g in set2 then remove b from set1                    *
*      4.  if b in set2 then remove g from set1                    *
*      - combine for common clauses                                *
*      - eliminate clauses in which set1 or set2 = 0               *
*  2.  Redundant or replicated input ports (or variables           *
*      derived from such input ports)                              *
*      - may not always hold , therefore, ask user                 *
*      - assumption:  x:g = y:g                                    *
*      - for x:(set1) <> y:(set2)                                  *
*        - separate into two possible clauses:                     *
*          1. if g in set1 then remove g from set2                 *
*          2. if g in set2 then remove g from set1                 *
*          - combine for common clauses                            *
*          - eliminate clauses in which set1 or set2 = 0           *
*          ( except for set1 = set2 = (g,b), there will            *
*            only be 1 possible clause)                            *
*  3.  Redundant or replicated input ports (or variables           *
*      derived such input ports)                                   *
*      - may not always hold , therefore, ask user                 *
*      - assumption:  x:b = y:b                                    *
*      - for x:(set1) <> y:(set2)  conditions                      *
*       - separate into two possible clauses:                      *
*          1. if b in set1 then remove b from set2                 *
*          2. if b in set2 then remove b from set1                 *
*          - combine for common clauses                            *
*          - eliminate clauses in which set1 or set2 = 0           *
*          ( except for set1 = set2 = (g,b), there will            *
*            only be 1 possible clause)                            *
*   4.  If n in set1 or set2 for all boolean comparators           *
* (<,>,=,<=,>=,<>), remove n from set1,set2                        *
*      - if set1 or set2 = 0 then remove clause                    *
*      - add clause:  n | x(i):n FA x(i):(set includes n)          *
*   5.  Interactively Verified Assumption:  (Phase 3)              *
*      (used last in propagation)                                  *
*       - hence try to eliminate characteristics                   *
*      RULE: if x:(set1) <rel> y:(set2) then ask:                  *
*        NEVER_TRUE (x:c = y:d) for all c,d in set1, set2          *
*          - for each T answer received by user:                   *
*          - separate into two possible clauses:                   *
*          1. if c in set1 then remove d from set2                 *
*          2. if d in set2 then remove c from set1                 *
*          - combine for common clauses                            *
*          - eliminate clauses in which set1 or set2 = 0           *
*****************************************************************
```

PHASE 2:

B. INSTANTIATE FOR EACH POSSIBLE OUTPUT CHARACTERISTIC:

```
*********************************************************
*  INSTANTIATION RULES:                                 *
*  1.  for every occurrence of VAR , sub VAR:(c)        *
*        -   perform propagation on all relations       *
*            effected by the substitutions in order     *
*            listed in substitutions.                   *
*  QUANTIFIER RULE:  C is the place where VAR was       *
*    substituted in step 1                              *
*  1.  FA var <cond> C  -->                             *
*        - eliminate quantifier (fully quantified)      *
*********************************************************
```

PROCEDURAL SPECIFICATION: find element to left of |:
(* solve for each characteristic possibilities for output
(* reduce resultant expression to right of |

1. if variable on left:
   for each characteristic associated with that variable
   a. instantiate for all occurrences of variable
   b. propagate
   c. add var:(char) as data condition on
      right of |
2. if function on left;
   a. use function propagation rules to define
      characteristics of output of function
   b. apply propagation of characteristics to all
      parameters.
   c. add var:(char) as data condition on
      right of | for all variables in function
   d. take out quantifiers that are now fully qualified
      - see QUANTIFIER RULE above
3. if single characteristic on left: done

END : FOR EACH FUNCTION DOMAIN

```
PHASE 3:

C.  after all domains in sentence have been analyzed:
    1.  combine for common output characteristics
    2.  Detect overlaps BETWEEN output characteristics
        FOR EACH output characteristic group
          (* check each pair of output characteristics *)
          For i = 1 to OC where OC is number of output characteristics
             for j = i+1 to OC
                check (i,j)  where i,j are unique output characteristics
             end
          end;
          CHECK (i,j):
             (* check each domain in i against each domain in j *)
             for k = 1 to n(i)  where k,l are clauses in i,j resp.
                for l = 1 to n(j)
                   paircheck(i.k,j.l)
                end;
             end;
           end CHECK;

           PAIRCHECK (i.k,j.l)
              (* check input characteristic for input x in i.k
                 against input characteristic for input x in j.l
                 do for all input variables.  If there is an
                 overlap, then call correct_conflict  *)
              for x = 1 to n  where n is number of inputs
                  if (INTERSECTION i.k(x) j.l(x)) = 0)
                      exit (* found difference *)
                  if NOT((INTERSECTION i.k(x) j.l(x)) = 0)
                      add x to conflict list;
              end for;
              (* if reaches here, then there was an overlap of input
                 characteristics *)

                 CORRECT_CONFLICT(i.k j.l)
           end PAIRCHECK;
```

PHASE 3: (con't)

```
CORRECT_CONFLICT (i.k j.l)
(* there is no input variable whose characteristics are
   mutually exclusive in the separate domains.  Therefore
   either the domains are identical for the input
   characteristics or one domain is a proper subset of the
   other domain *)

    1. for all conditions in i,j:
       apply rule 5 of CONDITION PROPAGATION RULES

       - if any reductions are made, perform
         paircheck again to see if overlap remains

    2. If no action taken in 1 or if overlaps persist:

       if any input variables on conflict list are not
       identical with the other domain, then either

       Heuristic: if the domains are for g output and b
       output, the system shift the overlap to the bad
       output. (ask user)

       - use for variable overlaps of b

       Or ask the user if the overlap could be attributed to
       one output characteristic only:  if the user answers
       no, then go to 3:

    3. If no action taken in #2:

       allow the overlap to remain and prompt the user to
       specify a percentage likelihood of occurrence for each
       domain.
END CORRECT_CONFLICT;
```

PHASE 3:(con't)

3. Detect and correct overlaps WITHIN each output definition
   (* this is done after #2 so that the resulting output
      definition for the component would be complete. *)

```
For i = 1 to C where C is number of clauses for this output
                characteristic
   for j = i+1 to C
       PAIRCHECK (i,j)  where i,j are unique claused for
                        this output characteristics
   end
end;

PAIRCHECK (i,j)
    (* check input characteristic for input x in i
       against input characteristic for input x in j
       do for all input variables.  If there is an
       overlap, then call correct_conflict_within  *)
    for x = 1 to n  where n is number of inputs
        if (INTERSECTION i(x) j(x)) = 0)
            exit (* found difference *)
        if NOT((INTERSECTION i(x) j(x)) = 0)
            add x to conflict list;
    end for;
    (* if reaches here, then there was an overlap of input
       characteristicsa *)

       CORRECT_CONFLICK_WITHIN(i,j);
 end PAIRCHECK;

·CORRECT_CONFLICK_WITHIN (i j)
 (* there is no input variable whose characteristics are
    mutually exclusive between domains.  Therefore either
    the domains are identical for the input characteristics
    or one domain is a proper subset of the other domain *)

    - choose the domain with that is the superset of the pair

    (* guarenteed not to overlap with any other output
       characteristic definition AND resulting output
       definition will be defined on all input
       characteristics.  *)
```

3. eliminate from each resulting output characteristic definition data characteristics that involve full enumeration of characteristic possibilities

4. FOR EACH CONDITION WITHIN EACH DOMAIN

   1. Separate conditions from characteristics
      (e.g. x:b and z:b and x:b > z:b)
   2. Eliminate reference to intermediate characteristics.

```
(* definition of intermediate variable:  any
(* data variable that is not the output variable
(* being analyzed and is not an input variable
```

APPENDIX C: MANUAL TRACE OF VOTED REDUNDANT PROCESSORS

The following is a trace of the algorithms for the Reliability model generator for the example system shown in figure C.1. The Building Blocks Definition for the system is shown in figure C.2. This is the same BBD as was illustrated in figure 7 and discussed in section 3.1.2.2. The only difference was in the naming of the input and output variables.

The System Definition is shown in figure C.3. Most of the SYSD may be generated automatically by the system based on the user specifying the highest level component to be analyzed (SYSTEM in example 1) and specifying failure modes for the lowest level components. However, some connections between components are not specified in the BBD, and therefore, need to be entered by the user. This is implemented as an interactive process between the system and the user. Figure C.2 indicates two connections (denoted by *1* and *3*) that are not specified in the BBD. Finally, if a component is redundant, the user is prompted to enter the redundancy level (denoted by *2* in figure C.2).

The desired output of the system is shown in figure C.4. This trace was performed according to the algorithms as they existed in September, 1987. Changes to these algorithms are itemized in appendix A. Despite the changes, the algorithms of appendix B retain the numerical cross reference that is used to identify the steps in the trace.

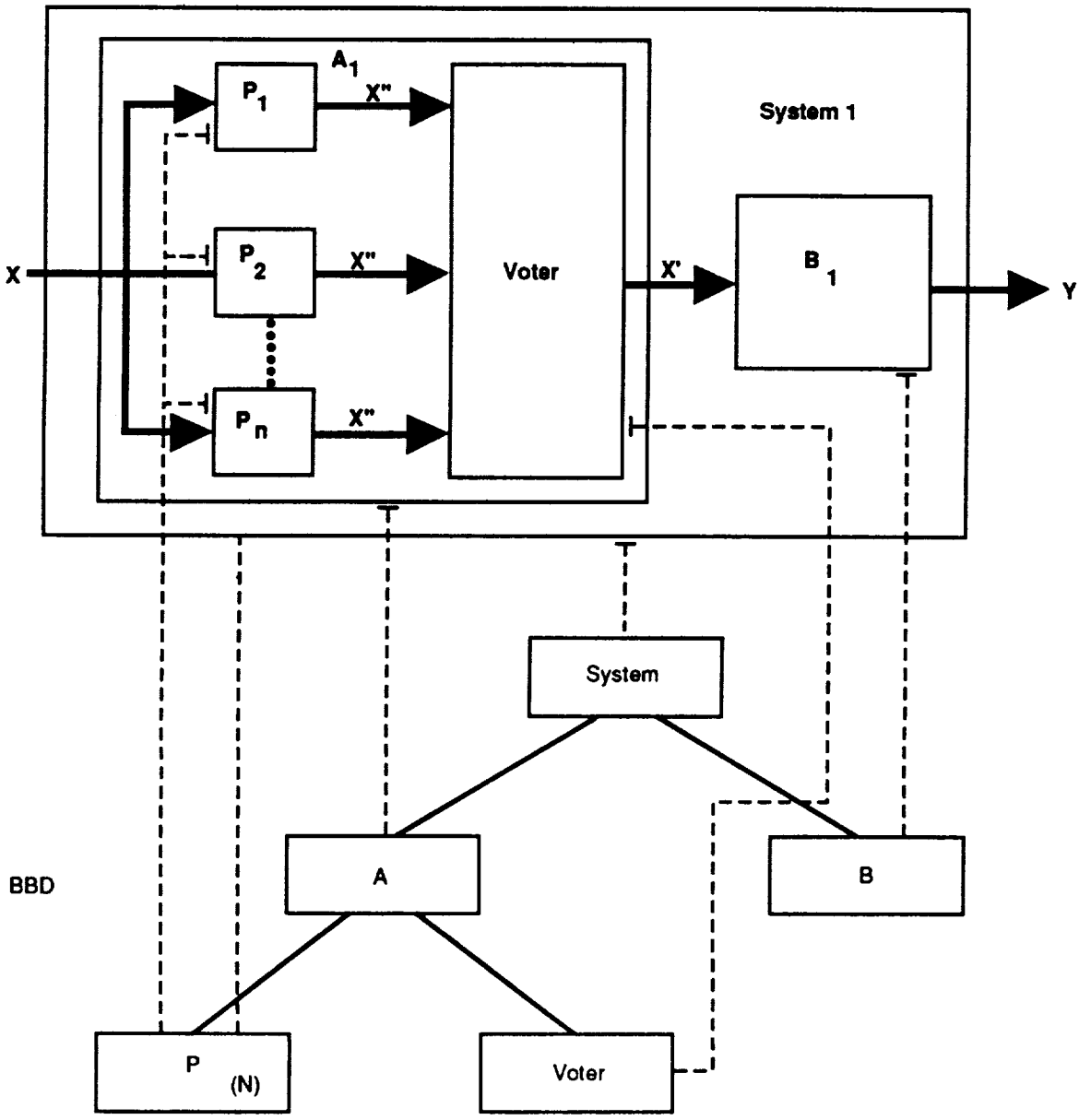


Figure C1. Voted Redundant Processor Example

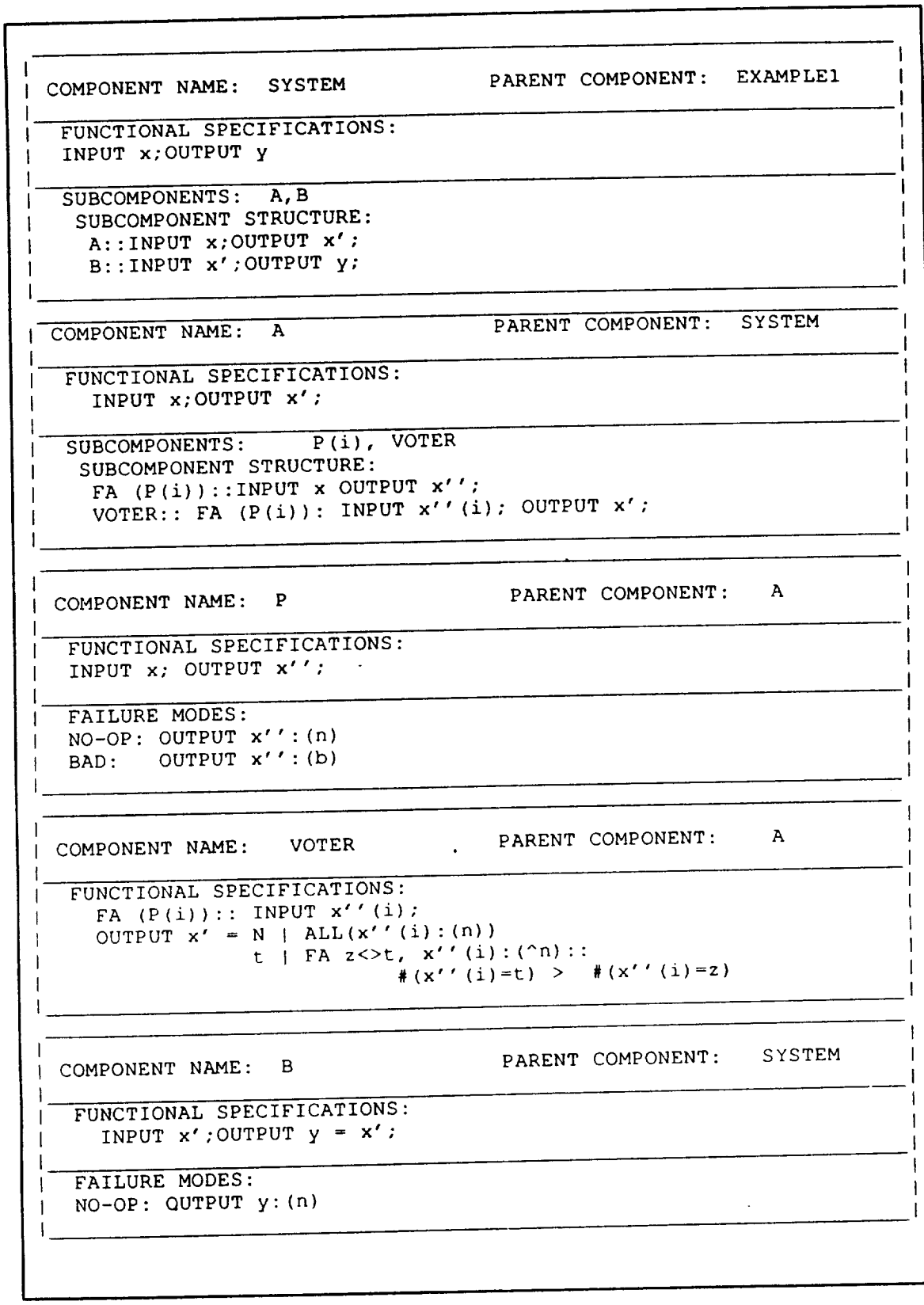


Figure C2. BBD for Example 1

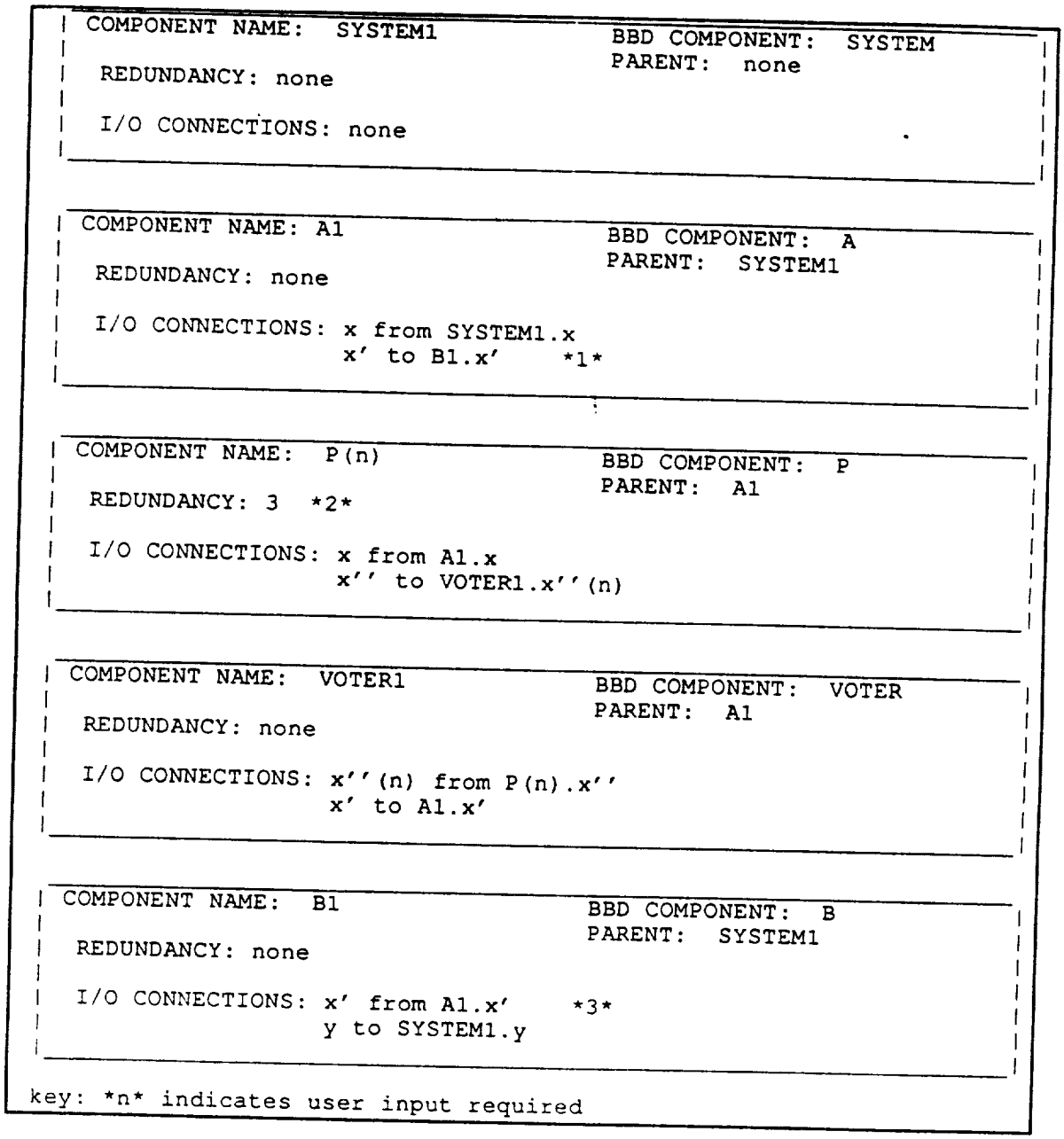


Figure C3. SYSD for Example 1

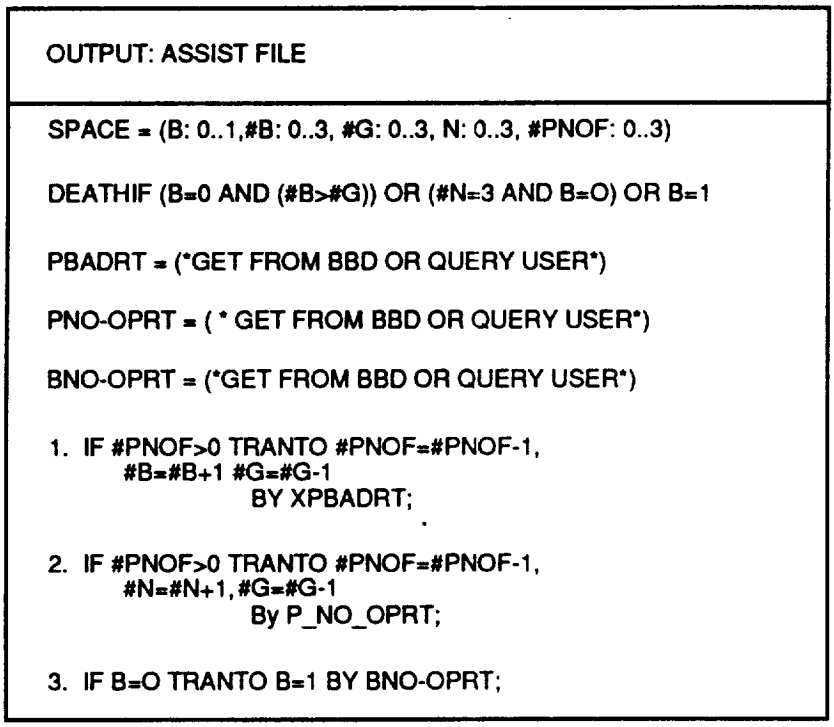


Figure C4. Example 1 Output

THIS IS A TRACE OF EXAMPLE 1 FOR ALL PROCESSES EXCEPT PROCESS3 AND PROCESS 2X WHICH ARE REFERENCED HERE WHEN INVOKED, BUT THE TRACES FOR THEIR ALGORITHMS ARE CONTAINED IN OTHER FILES. THE DOUBLE LINED ****** SEPARATE THE COMPONENT DOMAINS. THE ####### BOX ILLUSTRATES THE INFORMATION PASSED FROM THE PARENT COMPONENT TO ITS SUBCOMPONENTS.

EACH NUMBERED STEP INDICATES THE CONTROL FLOW OF THE PROCESSES FOR THIS EXAMPLE. THEY ARE NOT LISTED IN NUMERIC ORDER. RATHER, THE STEPS FOR EACH COMPONENT ARE LISTED TOGETHER TO ILLUSTRATE THE RECURSIVE TREE-LIKE BEHAVIOR OF THE ALGORITHMS. WHEN ONE COMPONENT CALLS ITS SUBCOMPONENT OR WHEN A SUBCOMPONENT PROCESS RETURNS TO ITS PARENT, THE STEP NUMBER IS GIVEN TO INDICATE WHAT TRACE STEP IS NEXT. THIS IS REPRESENTED BY A

FOR EACH STEP, THE PROCESS BEING INVOKED IS LISTED, AND EACH SUBSEQUENT NUMBER (OR LETTER) REFERENCES THE STEP IN ALGORITHM FOR THAT PROCESS.

```
**********************************************************************
**********************************************************************
1.  INITIALIZER:

    -highest level component is SYSTEM1
    -fill in table: history, outside effects=none
    ******************************************************
    ######################################################
    # TRACE TABLE:  SYSTEM1
    # function:              history        outside effects
    # input x;output y         none              none
    ######################################################
    -pass control to SYSTEM1

**********************************************************************
**********************************************************************

2.  IN SYSTEM 1:  not lowest level
    PROCESS 1:
    1.  set up trace tables for components A and B
    ######################################################
    # TRACE TABLE:  A
    # function               history        outside effects
    # input x; output x'       none              none
    ######################################################
    # TRACE TABLE:  B
    # function               history        outside effects
    # input x'; output y       none              ----
    ######################################################
    2.  set up priority Q (first-in-first-out)
    3.  call Process 2

    PROCESS 2:  take top element off list (A) and instantiate (3)

11.  IN SYSTEM1:
     PROCESS 2:
     1.  update external effects for component B
     2.  instantiate next subcomponent - B   pass to 12
         - returned output definitions and transitions for B
     3.  no more components - pass control to process 3 (16)

**********************************************************************

16.  PROCESS 3 in SYSTEM1:
TRANSITIONS:
GIVEN:
  FOR A:
       TRANSITIONS:
           1.  IF PiNOF TRANTO PiBAD,#(x''i:(B)=#(x''i:(B))+1,
                        #(x''i:(G)=#(x''i:(G))-1  BY P_BADRT

           2.  IF PiNOF TRANTO PiNO-OP,#(x''i:(N)=#(x''i:(N))+1,
                        #(x''i:(G)=#(x''i:(G))-1  BY P_NO-OPRT
```

```
          OUTPUT DEFINITION: for x'
             OUTPUT B IF #(X":(B)) > #(X":(G))
             OUTPUT N IF #(X":(N))=3)
             OUTPUT G IF #(X":(G)) > #(X":(B))

    FOR B:
        TRANSITIONS:
                    1.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

        OUTPUT DEFINITION:
             OUTPUT G IF x':(g) AND BNOF
             OUTPUT B IF x':(b) AND BNOF
             OUTPUT N IF (x':(n) AND BNOF) OR BNO-OP

RESULTS:
        OUTPUT DEFINITION:
             OUTPUT B IF BNOF AND (#(x''i:(b)) > #(x''i(g)))
             OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))
             OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP

        TRANSITIONS:
             1.  IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
                      #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
             2.  IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1,
                      #(X''i:(G)=#(X''i:(G))-1  BY P_NO_OPRT
             3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

    - see exam1.proc3.16.abs for description of this step
**********************************************************************
**********************************************************************
3.  IN COMPONENT A:  not lowest level
    PROCESS 1:
    1.  set up trace tables for components P and VOTER
    ############################################################
    # TRACE TABLE:   P(i)
    # function              history       outside effects
    # input x; output x'';    none            none
    ############################################################
    # TRACE TABLE:   VOTER
    # function              history       outside effects
    # input x'';output x'      none            ----
    ############################################################
    2.  set up priority Q (first-in-first-out - rules 1 and 3)
    3.  call Process 2

    PROCESS 2:  take top element off list (P) and instantiate (4)

7.  IN COMPONENT A:
    PROCESS 2:
    1.  )update other effects for VOTER -  rule 1
    ############################################################
    # TRACE TABLE:   P(i)
    # function              history       outside effects
    # input x; output x'';    none            none
```

```
    ############################################################
    # TRACE TABLE:   VOTER
    # function                history          outside effects
    # input x'';output x'       none              x'(i):(g,b,n)
    ############################################################
    2.  take top element off list (VOTER) and instantiate (8)

9.  IN COMPONENT A:
    PROCESS2:
    - finished with all subcomponents, pass to PROCESS3


10. COMPONENT A: PROCESS3
    GIVEN P's    OUTPUT N IF PiNO_OP
                 OUTPUT B IF PiBAD
                 OUTPUT G IF PiNOF

      VOTER'S    OUTPUT B IF #(x''i:(B)) > #(x''i:(G))
                 OUTPUT N IF ALL x''i:(N)
                 OUTPUT G IF #(x''i:(G)) > #(x''i:(B))
    ************************************************************
    * RESULT:
    * OUTPUT DEFINITION:  OUTPUT x':
    *            N IF #((x''i:(n))=3
    *                          G IF #(x''i:(g)) > #(x''i:(b))
    *                          B IF #(x''i:(b)) > #(x''i:(g))
    * TRANSITIONS:
    * 1. IF PiNOF TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
    *               #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
    * 2. IF PiNOF TRANTO PiNO_OP, #(X''i:(N))=#(X''i:(N)+1),
    *             #(X''i:(G))=#(X''i:(G))-1  BY P_NO_OPRT
    ************************************************************
  - see exam1.proc3.10.abs for abstract description of this
    step


**********************************************************************
**********************************************************************
4.  IN COMPONENT P:  lowest level
    PROCESS 1x:
    1.  create transitions for each fault possibility
    2.  substitute into functional definition:
    3.  Propagate:
              two results:
                  1.  input x; output x'':(n)  from fault NO-OP
                  2.  input x; output x'':(b)  from fault BAD
    4.  no history to analyze
    5.  output characteristics defined - don't call process 2x


5.
6.  PROCESS 3x:  Define output definition for each output characteristic
       OUTPUT (output characteristic)
                   IF (function defined in 2x) and (fault state)
```

```
      OUTPUT N IF () AND P_NO_OP
      - simplify: OUTPUT N IF P_NO_OP
      OUTPUT B IF () AND P_BAD
      - simplify: OUTPUT B IF P_BAD
      add in no fault state:  OUTPUT G IF P_NOF
   return to parent (A)  (7)


**************************************************************************
**************************************************************************
8.  IN COMPONENT VOTER:  lowest level
    PROCESS1X:
      - no faults to model (according to BBD) so only normal
        behavior to consider with external effects given

    PROCESS 2x:  -->  FA (P(i)):  input x''i FROM P(i)
                      output x' --> N | all x''i:(n)
                                      | FA z<>x', FA x''i:(^n):
                                            #(x''i=x') > #(x''i=z)
     FOR TRACE SEE PROCESS2X.VOTER


     RESULT:
         x':(n)  | all(x''i:(n)
         x':(g)  | #(x''i:(g)) > #(x''i:(b))
         x':(b)  | #(x''i:(b)) > #(x''i:(g))

     return to A  (7)

**************************************************************************
**************************************************************************
12  IN B: lowest level
    PROCESS 1x:
      1.  create transitions for each fault possibility
          IF BNOF TRANTO BNO-OP BY BNO-OPRT
      2.  substitute into functional definition:
          input x'; OUTPUT y:(n)
      3.  Propagate: no propagation necessary
      4.  no history to analyze
      5.  output characteristic defined: don't call process2x for
          NO_OP fault
      6.  call process2x for NOF state and x':(g,b,n) - goto 14
           - return from 14
    call process3x after all fault cases have been analyzed  - goto 15
13

14 PROCESS 2x in B for normal behavior
   GIVEN input x';output y from x'
         x':(g,b,n)

   UNSPECIFIED FUNCTION:
   y = O(x':(g,b,n))
   1.  for characteristic x':g
```

```
      - apply propagation rule: y=o(g) --> g
        OUTPUT y:(g) IF x':(g)
   2. for characteristic x':b
      - apply propagation rule:      y = O(b) --> b
        OUTPUT y:(b) IF x':(b)
   3. for characteristic x':n
      - apply propagation rule:  y = O(n) --> n
        OUTPUT y:(n) IF x':(n)

   no combining of common characteristics

15.  PROCESS 3x: Define output definition for each output characteristic
       OUTPUT (output characteristic)
                      IF (function define in 2x) and (fault state)

       OUTPUT Y:(N) IF () AND BNO-OP
                    OR x':(n) AND BNOF

       OUTPUT Y:(g) IF x':(g) AND BNO-OF

       OUTPUT Y:(b) IF x':(b) AND BNO-OF

     return to parent (system1 in step 11

*****************************************************************************
*****************************************************************************
PROCESS 4:
GIVEN:
  OUTPUT DEFINITION:
     OUTPUT B IF BNOF AND (#(x''i:(b)) > #(x''i(g)))
     OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))
     OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP

  TRANSITIONS:
     1.  IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
                         #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT

     2.  IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1,
                         #(X''i:(G)=#(X''i:(G))-1  BY P_NO_OPRT

     3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

1. APPLY MODEL REDUCTION TECHNIQUES:

     STEP 1:  find superfluous variables
        PiBAD, PiNO_OP

     STEP 2: combines superfluous variables for each component state
             and for each data variable
        PiANYOF(BAD,NO-OP)
     Substitute:
     1.  IF PNOF TRANTO P_ANYOF(BAD,NO-OP),#(x''i:(B)=#(x''i:(B))+1,
                         #(x''i:(G)=#(x''i:(G))-1  BY P_BADRT
```

```
   2.  IF PNOF TRANTO P_ANYOF(BAD,NO-OP),#(x''i:(N)=#(x''i:(N))+1,
                          #(x''i:(G)=#(x''i:(G))-1  BY P_NO_OPRT
   3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

   STEP 3: heuristic d: for redundant components:
   1.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#(x''i:(B)=#(x''i:(B))+1,
                         #(x''i:(G)=#(x''i:(G))-1  BY #PNOF x P_BADRT

   2.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#(x''i:(N)=#(x''i:(N))+1,
                         #(x''i:(G)=#(x''i:(G))-1  BY #PNOF x P_NO_OPRT
   3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

   NOTE:  this heuristic did not recognize that the #xi:g variable
          could be used for the same purpose as #PiNOF

 FINAL MODEL:
 OUTPUT DEFINITION:

 OUTPUT DEFINITION:
    OUTPUT B IF BNOF AND (#(x''i:(b)) > #(x''i(g)))
    OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))
    OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP

 TRANSITIONS:
   1.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#(x''i:(B)=#(x''i:(B))+1,
                         #(x''i:(G)=#(x''i:(G))-1  BY #PNOF x P_BADRT

   2.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#(x''i:(N)=#(x''i:(N))+1,
                         #(x''i:(G)=#(x''i:(G))-1  BY #PNOF x P_NO_OPRT
   3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

 TRANSLATING INTO ASSIST:
 The reliability question was: R(y:(b) or Y:(n))
 Therefore the death state would be:

 DEATH IF:{ BNOF AND (#(x''i:(b)) > #(x''i(g))) }
                    (* output definition for Y:(b) *)
          OR (#(x''i:(n))=3) AND BNOF) OR BNO-OP  }
                    (* output definition for Y:(n) *)
```

The variables will be represented as follows:

1. a state vector element for each component state is created. The number of values possible for the element will equal the number of states for the component with value 0 being the NOF state: For redundant components, the SYSD will determine the number of components: (* example:(SYSD say P(3) *) in exam1.

```
   B[0..1]     (* 0=BNOF, 1=BNO-OP *)
```

2. For variables #(x''i:(b)), #(x''i(g))), and #(x''i:(N)), a integer variable for each will be set up.

```
   #B: integer
```

#G: integer
#N: integer
#PNOF:integer

3. other variables will be represented in the same manner as in 1. A state vector element for each variable will be created, and the number of values possible for the element will equal th number of values or characteristics for the variable:
   - none in exam1

The transitions and death state will be changed to reflect the representation:

```
DEATH IF:{ B=0  AND (#B > #G ) }
          OR (#N=3 AND B=0 )
          OR B=1


 TRANSITIONS:
   1.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#B=#B+1,  #G=#G-1
                         BY #PNOF x PBADRT

   2.  IF #PNOF>0 TRANTO #PNOF=#PNOF-1,#N=#N+1,  #G=#G-1
                         BY #PNOF x P_NO_OPRT
   3.  IF B=0  TRANTO B=1 BY BNO-OPRT
```

This is an illustration of the reasoning involved in process2x for the voter component in exam1.

NOMENCLATURE:

```
<>   : not equal
FA   : for all
#    : number
-->  : function definition follows
|    : domain separator.
;    : sentence separator
^    : NOT
```

INTERNAL FUNCTIONAL REPRESENTATION (BBD) FOR VOTER:

```
FA (P(i)):
   input x''i FROM P(i);
   OUTPUT x' --> N | (ALL x''i:(n))
                   | FA z <> x', FA x''i:(^n):
                        #(x''i=x') > #(x''i=z)
```

READS: for all components P, input x''i from P. output x' which is defined by the following domains: x' has the characteristic N (nil) is all x''i are nil. Otherwise, x' is defined as the value in which the number of x''i equal to x' is greater than the number of x''i not equal to x' (in other words, x' is the majority of the values for x''i) x''i characteristics that are nil are excluded in the analysis

NOTE: function is not defined for inputs in which no majority exists

PROBLEM: Given the above function and the following input conditions:
x''i:(g,b,n)

Define the characteristics of the output as a function of its inputs

RESULT OF PROCESS 2x:

```
OUTPUT x':(n) | all(x''i:(n))
OUTPUT x':(g) | #(x''i:(g)) > #(x''i:(b))
OUTPUT x':(b) | #(x''i:(b)) > #(x''i:(g))
```

DISCUSSION:

1. Process 1x has previously considered the effects of faults on the component function. Process 2x considers the convolution of function (from 1x) and inputs (good and faulted) on the component outputs.

2. The output definition produced by process2x must not include reference to any variables other than input variables. This includes variables such as intermediate variables or variables used for quantifiers (such as z in example1).

3. The output definitions defined in process2x must not overlap either within a characteristic definition or between characteristics.

   Example1: overlap within a output characteristic definition

   ```
   output g IF x:(g) OR
            IF x:(g) and x':(b)
   ```

   must be simplified to:

   ```
   output g if x:(g)
   ```

   Example2: overlap between output characteristic definitions

   ```
   output g if x:(g)
   output b if x:(g) OR x':(b)
   ```

   is not acceptable either because the output can not be g and b when x:(g) as the definition indicates.

PROCESS:

CONSIDER EACH SENTENCE (DELINEATED BY ;) SERIALLY

FIRST SENTENCE: FA (P(i)): input x''i FROM P(i);

input functions are not analyzed in process 2x since they will be used in the BBD to bind variable names from one component outputing the variable with another component referencing the same variable as input.

SECOND SENTENCE:

OUTPUT x' --> N | (ALL x''i:(n))
| FA z <> x', FA x''i:(n): #(x''i=x') > #(x''i=z)

CONSIDER EACH DOMAIN SEPARATELY: (a domain is delineated by a |)

**** FIRST DOMAIN: ****

OUTPUT x' --> N | (ALL x''i:(n))

The input characteristics for x''i:(g,b,n) need not be considered since it has been determined that x'':(n)

**** SECOND DOMAIN: ****

OUTPUT x' --> | FA z <> x', FA x''i:(^n): #(x''i=x') > #(x''i=z)

SUBSTITUTE INPUT CHARACTERISTICS INTO FUNCTION:

The quantifier FA x''i:(^n) limits that characteristic possibilities for x''i. The inputs characteristics for x''i are x''i:(g,b,n). Combining these quantifiers, the characteristics for x''i are (g,b).

Substitute each possible characteristic for x''i:(g,b) into the first domain for every occurrence of x''i:

OUTPUT x' --> | FA z <> x', FA x''i:(g,b):
#(x''i:(g,b)=x') > #(x''i:(g,b)=z)

SECOND DOMAIN AFTER SUBSTITUTING INPUT CHARACTERISTICS:

OUTPUT x' --> | FA z <> x', FA x''i:(g,b):
#(x''i:(g,b)=x') > #(x''i:(g,b)=z)

PROPAGATE INPUT CHARACTERISTICS THROUGH FUNCTION DEFINITION TO OUTPUT:

These substitutions are propagated to other variables for each occurrence of the input characteristic based on the functional specification. For example:

OCCURRENCE: #(x''i:(g,b)=x')

#(x''i:(g,b)=x') is a variable which specifies the number of x''i that equal x', and it also specifies that x''i can be g or b. Based on the equality relationship between x''i and x', the characteristics of (g,b) are propagated to x'.

PROPAGATED: #(x''i:(g,b)=x':(g,b))

OCCURRENCE: #(x''i:(g,b)=z) similarly will result in the

PROPAGATED: #(x''i:(g,b)=z:(g,b))

RESULT AFTER PROPAGATION:

OUTPUT x' --> | FA z:(g,b) <> x':(g,b),x''i:(g,b):
#(x''i:(g,b)=x':(g,b)) >
#(x''i:(g,b)=z:(g,b))

SECOND DOMAIN AFTER PROPAGATING INPUT CHARACTERISTIC TO OUTPUT:

OUTPUT x' --> | FA z:(g,b) <> x':(g,b),x''i:(g,b):
#(x''i:(g,b)=x':(g,b)) >
#(x''i:(g,b)=z:(g,b))

INSTANTIATE FOR EACH CHARACTERISTIC OF OUTPUT x':

FOR x':(g):

OUTPUT x':(g) --> | FA z:(g,b) <> x':(g),x''i:(g,b):
#(x''i:(g,b)=x':(g)) >
#(x''i:(g,b)=z:(g,b))

PROPAGATE CHARACTERISTIC x':(g) THROUGH FUNCTION DEFINITION:

As in the substitutions, the characteristic g for variable x' is propagated through to the other variables via the functional definition:

OCCURRENCE 1. FA z:(g,b) <> x':(g):

This quantifier states that z is not equal to x', and z has the character possibility of g and b but x' has only one possible characteristic, g. From this, it can be concluded that

x' PROPAGATED to z: FA z:(b) <> x':(g)

Once a new characteristic for z is defined, it should be substituted in for all other occurrences of z and the propagation/substitution cycle continues until all occurrences of all affected variables have been considered.

SUBSTITUTE z:(b) IN FOR ALL OCCURRENCES OF z:

OCCURRENCE 1: #(x''i:(g,b)=z:(b))

z PROPAGATED TO x''i: #(x''i:(b)=z:(b))

DO NOT SUBSTITUTE x''i:(b) IN FOR ALL OCCURRENCES OF x''i: since x''i is a variable that represents more than one component

```
PROPAGATE CHARACTERISTIC x':(g) THROUGH FUNCTION DEFINITION:

     OUTPUT x':(g)   -->   | FA z:(g,b) <> x':(g),x''i:(g,b):
                                  #(x''i:(g,b)=x':(g)) >
                                                  #(x''i:(g,b)=z:(g,b))


     OCCURRENCE 2: #(x''i:(g,b)=x':(g)):

            x' PROPAGATED TO x''i: #(x''i:(g)=y:(g))

            DO NOT SUBSTITUTE x''i:(b) IN FOR ALL
            OCCURRENCES OF x''i


RESULT AFTER PROPAGATING/SUBSTITUTION CYCLE FOR INSTANTIATION X':(g):


x':(g)  |  #(x''i:(g)=x':(g)) > #(x''i:(b)=z:(b))
```

INSTANTIATE FOR EACH CHARACTERISTIC OF OUTPUT x':

FOR x':(b):

OUTPUT x':(b) --> | FA z:(g,b) <> x':(b),x''i:(g,b):
#(x''i:(g,b)=x':(b)) >
#(x''i:(g,b)=z:(g,b))

PROPAGATE CHARACTERISTIC x':(b) THROUGH FUNCTION DEFINITION:

OCCURRENCE 1. FA z:(g,b) <> x':(b):

x' PROPAGATED to z: FA z:(g) <> x':(b)

SUBSTITUTE z:(g) IN FOR ALL OCCURRENCES OF z:

OCCURRENCE 1: #(x''i:(g,b)=z:(g))

z PROPAGATED TO x''i: #(x''i:(g)=z:(g))

DO NOT SUBSTITUTE x''i:(g) IN FOR ALL
OCCURRENCES OF x''i:

OCCURRENCE 2: #(x''i:(g,b)=x':(b)):

x' PROPAGATED TO x''i: #(x''i:(b)=x':(b))

DO NOT SUBSTITUTE x''i:(b) IN FOR ALL
OCCURRENCES OF x''i

RESULT AFTER PROPAGATING/SUBSTITUTION CYCLE FOR INSTANTIATION X':(g):

x':(b) | #(x''i:(b)=x':(b)) > #(x''i:(g)=z:(g))

Results for both instantiations of x' have created two domains:

```
x':(g)  |  #(x''i:(g)=x':(g))  >  #(x''i:(b)=z:(b))
x':(b)  |  #(x''i:(b)=x':(b))  >  #(x''i:(g)=z:(g))
```

ELIMINATE REFERENCE TO X' ON RIGHT SIDE OF |
The reference to x' on the right side of the | may now be eliminated since each new domain directly specifies the characteristic for x'

ELIMINATE REFERENCE TO Z
Further, recall from the discussion prior to this example that variables such as z should not appear in the final result. Note that the propagation/substitution cycle propagated the characteristic for z to any affected variables. Therefore, the fact that z:(b) or z:(g) is not needed in the function specification. and its reference may be eliminated.

FINAL RESULT FOR DOMAIN 2:

```
x':(g)  |  #(x''i:(g))  >  #(x''i:(b))
x':(b)  |  #(x''i:(b))  >  #(x''i:(g))
```

FINAL RESULT FOR ALL DOMAINS:

```
x':(n)  |  all(x''i:(n)
x':(g)  |  #(x''i:(g))  >  #(x''i:(b))
x':(b)  |  #(x''i:(b))  >  #(x''i:(g))
```

The following is an overview of the processing involved in example1 when process3 is invoked in component A. Details of the processing can be found in EXAM1.TRACE2.ABS. This example illustrates the reasoning behind the process flow (shown in small letters) and shows a procedural ALL CAPTITALS).

```
**************************************************************************
STEP 10 in exam1.trace.abs:  PROCESS 3 in component A:

COMPONENT A: PROCESS3
    GIVEN P's OUTPUT DEFINITION:
                OUTPUT N IF PiNO_OP
                OUTPUT B IF PiBAD
                OUTPUT G IF PiNOF
          P's TRANSITIONS:
                1.  IF PiNOF TRANTO PiBAD BY PBADRT
                2.  IF PiNOF TRANTO PiNO-OP BY PNO_OPRT

          VOTER'S DEFINITION:
                OUTPUT B IF #(x''i:(B)) > #(x''i:(G))
                OUTPUT N IF ALL(x''i:(N))
                OUTPUT G IF #(x''i:(G)) > #(x''i:(B))
          VOTER'S TRANSITIONS: none

DESIRED OUTPUT:

     OUTPUT DEFINITION:
          OUTPUT B IF #(x''i:(B)) > #(x''i:(G))   10
          OUTPUT N IF #(x''i:(N))=3)
          OUTPUT G IF #(x''i:(G)) > #(x''i:(B))

     TRANSITIONS:

          1.  IF PiNOF TRANTO PiBAD,#(x''i:(B)=#(x''i:(B))+1,
                        #(x''i:(G)=#(x''i:(G))-1   BY PBADRT

          2.  IF PiNOF TRANTO PiBAD,#(x''i:(N)=#(x''i:(N))+1,
                        #(x''i:(G)=#(x''i:(G))-1   BY PNO_OPRT

**************************************************************************
DISCUSSION:  This process is called once by every parent component (a
component that has subcomponents) to combine the output definition and
transitions of the subcomponents into an output definition for the
component.  (This is then 'given' to its parent component who repeats this
process for its subcomponents and so on until the top level component).
The top level output definition will be encoded into a state space model
and further model reduction techniques will be applied in process4.

The following criteria for the desired output (output definition and
transitions) of each instantiation of this process must be met:

     1.  The transitions and the output definition must reference the
         same variables.  Therefore, the process 3 must decide which
         variables are necessary to describe the state space and which
         variables can be eliminated.  The criteria for eliminating
         variables will be discussed later.

     2.  Changes to all affected variables must be reflected in a
         transition.  For example, transition #1 in P's transition
         list must be changed to reflect the change in #(x''i:(b))
```

and in #(x''i:(g)) because these variables are referenced in the output definition.

3. All transitions must be found. A transition will result from a component fault. In this case, the transition already exists (from process1x) and must be changed according to the two criteria discussed above. However, a transition may also be non-fault related, such as a transition that models the recovery from a component fault. Such a transition is not explicitly referenced in the individual subcomponent definitions and must be 'found' first and then changed.

The way in which the output definition and transitions for a parent component will be defined from the subcomponent output definitions and transitions is similar to the way in which the same task is accomplished manually. First, the subcomponent that produces the final output is analyzed. Its output definition and transitions are examined. This output definition references the input characteristics that contribute to the output. In order to determine the transitions that caused the input characteristics, the next subcomponent analyzed is the component that outputed the data that became input to the last component. This process is repeated (in this backward chaining manner) until the inputs to the parent component are reached or the cycle repeats itself (as in the case of a cyclical process). The following example shows this without referencing the procedural details of process3:

****

STEP 1: Look at the subcomponent that produces the final output: the voter.

VOTER'S DEFINITION:
OUTPUT B IF #(x''i:(B)) > #(x''i:(G))
OUTPUT N IF ALL x''i:(N)
OUTPUT G IF #(x''i:(G)) > #(x''i:(B))
VOTER'S TRANSITIONS: none

ANALYZE EACH POSSIBLE OUTPUT CHARACTERISTIC SEPARATELY:

1. OUTPUT B IF #(x''i:(B)) > #(x''i:(G)):

Stated as: the output is bad if the number of inputs x that are bad outnumber the number of inputs that are good. In order to find the transitions that effect the output being bad, find the transitions that contribute to and detract from the number of inputs being bad and the number of inputs being good.

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY

1. #(x''i:(B)) : find the transitions that affect this condition:

The transitions that contribute to the number of x inputs

being bad is directly related to the transitions that contribute to one x value being bad, so find transitions that contribute to x being bad:

FOR OPERATIONS SUCH AS #, THERE HAS TO BE SOME 'KNOWLEDGE' IN THE SYSTEM TO DETERMINE WHAT CONDITION TO ANALYZE (CALLED CONDITION OF INTEREST - COI) ++ 3 ++

x''i:(b): find transitions that affect this conditions:

Since this is an input to the voter, look at the P component definition that outputs x''i:(b):

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT x''i:(b) IF PiBAD

Again, in order to analyze the conditions under which P's output is bad, find the transitions that affect the condition PiBAD:

ANALYZE THE CONDITION (PiBAD) UNDER WHICH THE OUTPUT CHARACTERISTIC (B) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- PiBAD is a fault state of component P that results when P fails by fault BAD (as specified in the BBD definition of P). Therefore the transition that contributes to PiBAD is:

P's TRANSITIONS:
1. IF PiNOF TRANTO PiBAD BY PiBADRT

This transition contributes to PiBAD, but the COI is x''i:(b). Since P's output definition states that the output is b if PiBAD, we can state the following directly:

1. IF PiNOF TRANTO PiBAD, x''i:(b) BY PiBADRT

THE TRANSITION IS CHANGED TO REFLECT THE CHANGE TO THE CONDITION BEING ANALYZED (x''i:(b)) ++ 4 ++

Once a transition has been found that contributes to x''i:(b), change it to reflect an effect to #(x''i:(b)) which was the original condition being analyzed.

1. IF PiNOF TRANTO PiBAD, #(x''i:(B)=#(x''i:(b))+1
BY PBADRT

THE TRANSITION IS CHANGED TO REFLECT THE CHANGE
TO THE CONDITION BEING ANALYZED #(x''i:(b)) ++ 4 ++

2. #(x''i:(g)) : find the transitions that affect this condition:

As in the case for #(x''i:(b)), the transitions that contribute to the number of x inputs being good is directly related to the transitions that contribute to one x value being good, so find transitions that contribute to x being good:

FOR FUNCTIONS SUCH AS #, THERE HAS TO BE SOME 'KNOWLEDGE' IN THE SYSTEM TO DETERMINE WHAT CONDITION TO ANALYZE ++ 3 ++

x''i:(g): find transitions that affect this conditions:

Since this is an input to voter, look at P component definition that outputs x''i:(g):

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT x''i:(g) IF PiNOF

ANALYZE THE CONDITION (PiNOF) UNDER WHICH THE OUTPUT CHARACTERISTIC (G) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- PiNOF is a no fault state and there is no transition that leads to this state.

Since no transitions contribute to x''i:(g), there are no transitions that contribute to #(x''i:(g)). However, only transitions that contribute to #(x''i:(g)) have been analyzed. To determine transitions that detract from #(x''i:(g)), look at transitions that have already been defined as contributing to other conditions and determine if these transitions detract from #(x''i:(g)). Transition #1 does detract from #(x''i:(g)) and it must be changed to reflect this:

1. IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
#(X''i:(G)=#(X''i:(G))-1 BY P_BADRT

ALL PREVIOUSLY DEFINED TRANSITIONS THAT DETRACT FROM THE CURRENT CONDITION ARE CHANGED TO REFLECT THIS? THERE MUST BE A WAY FOR THE PROCESS 3 TO DETECT WHEN A PREVIOUSLY DEFINED TRANSITION DETRACTS FROM A CURRENT CONDITION ++ 5 ++

Therefore in analyzing the subconditions #(x''i:(b)) and #(x''i:(g)), we have found 1 transition to contribute to #(x''i:(b)) > #(x''i:(g)):

1. IF PiNOF TRANTO PiBAD,#(x''i:(B)=#(x''i:(b))+1
#(X''i:(G)=#(X''i:(G))-1 BY PBADRT

2. OUTPUT N IF ALL(X''i:(n)):

ANALYZE THE CONDITION UNDER WHICH THE OUTPUT CHARACTERISTIC CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- ALL(X''i:(n)): find the transitions that affect this condition

One way in which analysts represent the transition to a system state in which all x inputs are n is to keep track of the number of x inputs that are n and to determine when that number equals the total number of possible x values. Therefore, an analyst would change the condition ALL(X''i:(n)) to #(x''i:(n)) = 3) where 3 is the total number of x inputs as determined by the BBD specification. Therefore, the analyst tries to find the transitions that contributes to #(x''i:(n)):

FOR SOME OPERATIONS (SUCH AS THE OPERATION 'ALL') THE REPRESENTATION OF THE CONDITION IS CHANGED IN ORDER TO MODEL IT LATER. ++ 3 ++

- #(x''i:(n)): find the transitions that affect this condition

As in #(x''i:(b)) and #(x''i:(g)), find the transitions that contribute to x''i:(n) and then change any transitions to reflect a contribution to #(x''i:(n))

- x''i:(n): find the transitions that affect this condition:

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT N IF PiNO_OP

ANALYZE THE CONDITION (PiNO_OP) UNDER WHICH THE OUTPUT CHARACTERISTIC (N) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- PiNO_OP is a fault state of component P that results when P fails by fault NO_OP (as specified in the BBD definition of P). Therefore, the transition that contributes to PiNO_OP is:

P's TRANSITION:

2. IF PiNOF TRANTO PiNO_OP BY PiNO_OPRT .

THE TRANSITION IS CHANGED TO REFLECT THE CHANGE
TO THE CONDITION BEING ANALYZED (x''i:(n)) ++ 4 ++

2. IF PiNOF TRANTO PiNO_OP, x''i:(n) BY PiNO_OPRT

THE TRANSITION IS CHANGED TO REFLECT THE CHANGE TO THE CONDITION BEING ANALYZED #(x''i:(n))

2. IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1 BY PiNO_OPRT

- Again, look at previously defined transitions and determine whether or not they detract from #(x''i:(n)): transition #1 does not. But notice that the newly defined transition #2 detracts from #(x''i:(g)). Therefore this transition must be changed to reflect the affect on #(x''i:(g)):

2. IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1,
#(X''i:(G)=#(X''i:(G))-1,
BY PiNO_OPRT

NOT ONLY DOES PROCESS 3 HAVE TO CHECK ALL PREVIOUS TRANSITIONS FOR AN DETRACTORY EFFECT ON THE CURRENT CONDITION BEING ANALYZED, BUT MUST ALSO CHECK ALL CURRENTLY DEFINED TRANSITIONS AGAINST ANY PREVIOUSLY DEFINED CONDITIONS. ++ 5 ++

3. OUTPUT G IF #(X''i:(G)) > #(X''i:(B)) :

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY:

- #(X''i:(g)): find the transitions that affect this condition:
  - this condition has already been analyzed

- #(X''i:(b)): find the transitions that affect this condition:
  - this condition has already been analyzed

  - no more transitions found

The final output for this process is:

OUTPUT DEFINITION:
OUTPUT x':(b) IF #(X''i:(b)) > #(X''i:(g))
OUTPUT x':(n) IF #(X''i:(n))=3)
OUTPUT x':(g) IF #(X''i:(g)) > #(X''i:(b))

TRANSITIONS:

1. IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
#(X''i:(G)=#(X''i:(G))-1 BY PiBADRT

```
2.  IF PiNOF TRANTO PiBAD,#(X''i:(N)=#(X''i:(N))+1,
                          #(X''i:(G)=#(X''i:(G))-1
                                      BY PiBADRT
```

```
The following is an overview of the processing involved in example1 for
process3.  This example illustrates the reasoning behind the process flow
(shown in small letters) and then shows a procedural algorithm that could
be used to implement the reasoning (shown in ALL CAPITALS)

*************************************************************************
STEP 16 in exam1.trace.abs:  PROCESS 3 in component SYSTEM1:

COMPONENT SYSTEM: PROCESS3
GIVEN:
  FOR A:
       TRANSITIONS:
           1.  IF PiNOF TRANTO PiBAD,#(x''i:(B)=#(x''i:(B))+1,
                                     #(x''i:(G)=#(x''i:(G))-1  BY P_BADRT

           2.  IF PiNOF TRANTO PiBAD,#(x''i:(N)=#(x''i:(N))+1,
                                      #(x''i:(G)=#(x''i:(G))-1  BY P_BADRT

         OUTPUT DEFINITION:
            OUTPUT B IF #(x''i:(B)) > #(x''i:(G))
            OUTPUT N IF #(x''i:(N))=3)
            OUTPUT G IF #(x''i:(G)) > #(x''i:(B))

   FOR B:
       TRANSITIONS:
                   1.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

       OUTPUT DEFINITION:
            OUTPUT G IF x'(g) AND BNOF
            OUTPUT B IF x'(b) AND BNOF
            OUTPUT N IF (x'(n) AND BNOF) OR BNO-OP

DESIRED OUTPUT:

     OUTPUT DEFINITION:
          OUTPUT B IF BNOF AND (#(x''i:(B)) > #(x''i:(G)))
          OUTPUT N IF (BNOF AND (#(x''i:(N))=3)) OR BNO-OP
          OUTPUT G IF BNOF AND (#(x''i:(G)) > #(x''i:(B)))

     TRANSITIONS:
     1.  IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
                           #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT

     2.  IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1,
                         #(X''i:(G)=#(X''i:(G))-1  BY P_NO_OPRT

*************************************************************************
DISCUSSION:  See exam1.proc3.10.abs
             This step in example 1 will show how process 3 combines
             component models that had previously been defined for
             subcomponents.  In this example subcomponents A and B
             abstract models will be combined to form component SYSTEM1
             abstract model.  Previously process 3 was instantiated to
             form component A abstract model from components P and Voter.
```

****

STEP 1: Look at the subcomponent that produces the final output: component B.

COMPONENT B'S DEFINITION:
OUTPUT B IF x'(b) AND BNOF
OUTPUT G IF x'(g) AND BNOF
OUTPUT N IF (x'(n) AND BNOF) OR BNO-OP

B'S TRANSITIONS:
1. IF BNOF TRANTO BNO-OP BY BNO-OPRT

ANALYZE EACH POSSIBLE OUTPUT CHARACTERISTIC SEPARATELY:
1. OUTPUT B IF x':(b) AND BNOF

Stated as: the output is bad if the input x' is bad and if component B is not failed. In order to find the transitions that effect the output being bad, find the transitions that contribute to and detract from the input x' being bad and the transitions contributing and detracting from BNOF.

ANALYZE THE CONDITION x':(b) AND BNOF UNDER WHICH THE OUTPUT CHARACTERISTIC (B) CAN OCCUR
TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- x':(b) AND BNOF: find the transitions that affect these conditions:

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY

1. BNOF: find the transitions that affect this condition:

- BNOF is a no fault state and there is no transition that leads to this state.

2. x':(b): find the transitions that affect this condition:

Since this is an input to component B, look at component A's definition that outputs x':(b).

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

A's OUTPUT DEFINITION:
OUTPUT x':(b) IF #(x''i:(B)) > #(x''i:(G))

ANALYZE THE CONDITION (#(X''i:(B)) > #(X''i:(G))) UNDER WHICH THE OUTPUT CHARACTERISTIC (b) CAN OCCUR TO

DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- since component A is an intermediate level component, an abstract model has been defined for its output characteristics. Further the transitions that both contribute and detract from the output characteristics have been defined:
  1. IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_BADRT

  2. IF PiNOF TRANTO PiNO-OP,#(X''i:(N)=#(X''i:(N))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_NO_OPRT

- at this point we have:
  B: OUTPUT B IF x':(b) AND BNOF
  A: OUTPUT x':(b) IF #(X''i:(B)) > #(X''i:(G))
  and transitions:
  1. IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_BADRT

  2. IF PiNOF TRANTO PiNO-OP,#(X''i:(N)=#(X''i:(N))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_NO_OPRT

  that reference data variables of A's output definition: Therefore, change the output definition for B to

  B: OUTPUT B IF (#(X''i:(B)) > #(X''i:(G))) AND BNOF

  and eliminate the reference to x'.

  Note that here, the output definition is changed to reference the same variables as the transitions found whereas in other situations in exam1, the transitions were changed to reflect the variables in the output definition.

  WHEN A INPUT CHARACTERISTIC (x':(b)) HAS BEEN ANALYZED AND TRANSITIONS HAVE BEEN FOUND, EITHER THE TRANSITIONS WILL HAVE TO BE CHANGED TO REFLECT AN AFFECT ON THE INPUT CHARACTERISTIC OR THE OUTPUT DEFINITION BEING ANALYZED WILL HAVE TO BE MODIFIED TO REFERENCE THAT SAME VARIABLES AS THE TRANSITIONS REFERENCE. ++ 4 ++

- the result of analyzing x':(b):

  OUTPUT B IF (#(X''i:(B)) > #(X''i:(G))) AND BNOF
  1. IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_BADRT
  2. IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_NO_OPRT

ANALYZE EACH POSSIBLE OUTPUT CHARACTERISTIC SEPARATELY:
2. OUTPUT N IF (x'(n) AND BNOF) OR BNO-OP

ANALYZE THE CONDITION (x'(n) AND BNOF) OR BNO-OP UNDER WHICH

THE OUTPUT CHARACTERISTIC (n) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY

1. x'(n) AND BNOF - find the transitions that contribute to this condition

   through the same reasoning as in the analysis of OUTPUT B ..., the following transitions and output definition will be determined:

   OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP
   TRANSITIONS: same as above

2. BNO-OP - find the transitions that contribute to this condition

   - BNO_OP is a fault state of component B that results when B fails by fault NO_OP (as specified in the BBD definition of B). Therefore, the transition that contributes to BNO_OP is:

     B's TRANSITION:
       3. IF BNOF TRANTO BNO_OP BY BNO_OPRT

   WHEN A INPUT CHARACTERISTIC (x':(b)) HAS BEEN ANALYZED AND TRANSITIONS HAVE BEEN FOUND, EITHER THE TRANSITIONS WILL HAVE TO BE CHANGED TO REFLECT AN AFFECT ON THE INPUT CHARACTERISTIC OR THE OUTPUT DEFINITION BEING ANALYZED WILL HAVE TO BE MODIFIED TO REFERENCE THAT SAME VARIABLES AS THE TRANSITIONS REFERENCE. ++ 4 ++

   Since the transition (3) and the output definition both refer to the same variable (BNO-OP), no changes to the transition or the output definition need to be made.

   CHECK ALL PREVIOUS TRANSITIONS FOR AN DETRACTORY EFFECT ON THE CURRENT CONDITION BEING ANALYZED AND CHECK ALL CURRENTLY DEFINED TRANSITIONS AGAINST ANY PREVIOUSLY DEFINED CONDITIONS. ++ 5 ++

   - no such interrelationship

3. OUTPUT G IF x'(g) AND BNOF:

   DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY:

   - x'(g): find the transitions that affect this condition:
     - returns same transitions as in x':(b)

```
        - changes output definition to
              OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))

    - BNOF: find the transitions that affect this condition:
          -  this condition has already been analyzed

          -  no more transitions found

The final output for this process is:

OUTPUT DEFINITION:
     OUTPUT B IF BNOF AND (#(x''i:(b)) > #(x''i(g)))
     OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))
     OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP

TRANSITIONS:
     1.  IF PiNOF TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1,
                              #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
     2.  IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1,
                          #(X''i:(G)=#(X''i:(G))-1  BY P_NO_OPRT
     3.  IF BNOF TRANTO BNO-OP BY BNO-OPRT
```

APPENDIX D: MANUAL TRACE OF VOTED REDUNDANT PROCESSORS WITH REDUNDANCY MANAGEMENT

The following is a trace of the algorithms for the Reliability model generator for the example system shown in figure D.1. The Building Blocks Definition for the system is shown in figure D.2. A feedback signal, r, is added between the voter and the processors, P, such that if an input signal to the VOTER disagrees with the majority of input signals, the VOTER outputs a "disable" signal, r, to the corresponding processor. The processors, P, behave as in example 1, except that upon receiving a disable signal from the voter "r=1", the processors output nothing (nil). Also, the failure mode, BAD, for the processor is changed so that it outputs corrupted data only if r is not 1. See figure C.2 for the changes in the BBD.

The System Definition is shown in figure D.3. The desired output of the system is shown in figure D.4. Most of the SYSD may be generated automatically by the system based on the user specifying the highest level component to be analyzed (SYSTEM in example 1) and specifying failure modes for the lowest level components. However, some connections between components are not specified in the BBD, and therefore, need to be entered by the user. This is implemented as an interactive process between the system and the user. Figure D.2 indicates two connections (denoted by *1* and *3*) that are not specified in the BBD. Finally, if a component is redundant, the user is prompted to enter the redundancy level (denoted by *2* in figure D.2).

These trace was performed according to the algorithms as they existed in September, 1987. Changes to these algorithms are itemized in appendix A. Despite the changes, the algorithms of appendix B retain the numerical cross reference that is used to identify the steps in the trace.

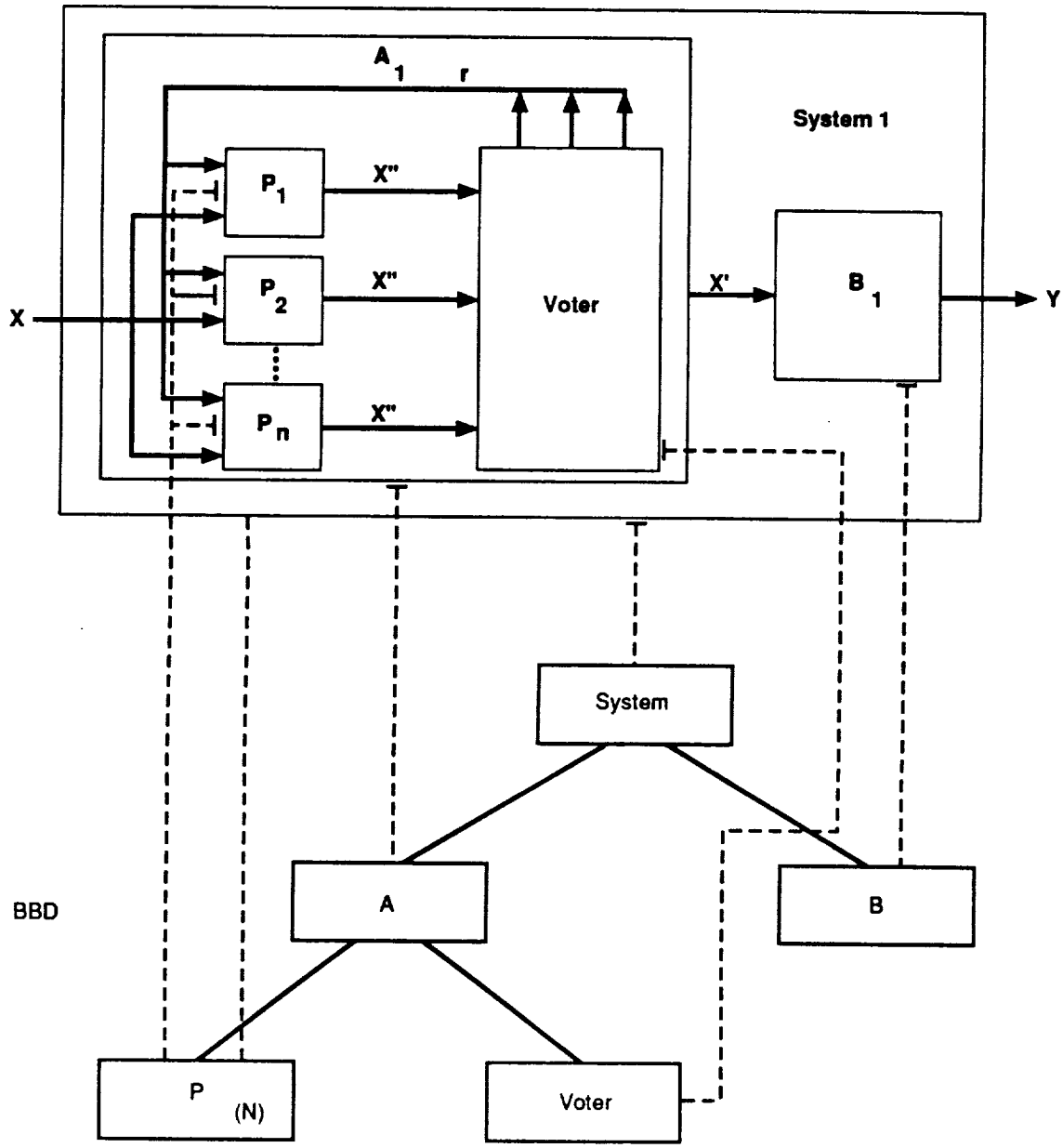


Figure D1. Voted Redundant Processor Example

```
COMPONENT NAME:  SYSTEM                    PARENT COMPONENT:  EXAMPLE1

 FUNCTIONAL SPECIFICATIONS:
 INPUT x;OUTPUT y

 SUBCOMPONENTS:  A,B
  SUBCOMPONENT STRUCTURE:
   A::INPUT x;OUTPUT x';
   B::INPUT x';OUTPUT y;

COMPONENT NAME:  A                         PARENT COMPONENT:  SYSTEM

 FUNCTIONAL SPECIFICATIONS:
   INPUT x;OUTPUT x';

 SUBCOMPONENTS:      P(i), VOTER
  SUBCOMPONENT STRUCTURE:
   FA (P(i))::(INPUT x,r(i);  OUTPUT x'';
   VOTER:: FA (P(i)): INPUT x''(i); OUTPUT x';
                              OUTPUT  r(i) to P(i);

COMPONENT NAME:  P                         PARENT COMPONENT:  A

 FUNCTIONAL SPECIFICATIONS:
 INPUT x,r;
 OUTPUT x'' = x | r=0;
              :n | r=1;

 FAILURE MODES:
 NO-OP: OUTPUT x'':(n)
 BAD:   OUTPUT x'':(b) | r=0

COMPONENT NAME:  VOTER                     PARENT COMPONENT:  A

 FUNCTIONAL SPECIFICATIONS:
   FA (P(i)):: INPUT x''(i);
   OUTPUT x' =  N | ALL(x''(i):(n))
                t | FA z<>t, x''(i):(^n)::
                    #(x''(i)=t) >  #(x''(i)=z)
   OUTPUT r(i) =  0 | x''(i) = x'
                  1 | x''(i) <> x';

COMPONENT NAME:  B                         PARENT COMPONENT:  SYSTEM

 FUNCTIONAL SPECIFICATIONS:
   INPUT x';OUTPUT y = x';

 FAILURE MODES:
 NO-OP: OUTPUT y:(n)
```

Figure D2. BBD for Example 2

```
COMPONENT NAME:  SYSTEM1                BBD COMPONENT:  SYSTEM
                                        PARENT:  none
 REDUNDANCY: none

 I/O CONNECTIONS: none

COMPONENT NAME: A1                      BBD COMPONENT:  A
                                        PARENT:  SYSTEM1
 REDUNDANCY: none

 I/O CONNECTIONS: x from SYSTEM1.x
                  x' to B1.x'     *1*

COMPONENT NAME:  P(n)                   BBD COMPONENT:  P
                                        PARENT:  A1
 REDUNDANCY: 3  *2*

 I/O CONNECTIONS: x from A1.x
                  r from VOTER.r(n)
                  x'' to VOTER1.x''(n)

COMPONENT NAME:  VOTER1                 BBD COMPONENT:  VOTER
                                        PARENT:  A1
 REDUNDANCY: none

 I/O CONNECTIONS: x''(n) from P(n).x''
                  x' to A1.x'
                  r(i) to A1.r

COMPONENT NAME:  B1                     BBD COMPONENT:  B
                                        PARENT:  SYSTEM1
 REDUNDANCY: none

 I/O CONNECTIONS: x' from A1.x'     *3*
                  y to SYSTEM1.y

key: *n* indicates user input required
```

Figure D3. SYSD for Example 1

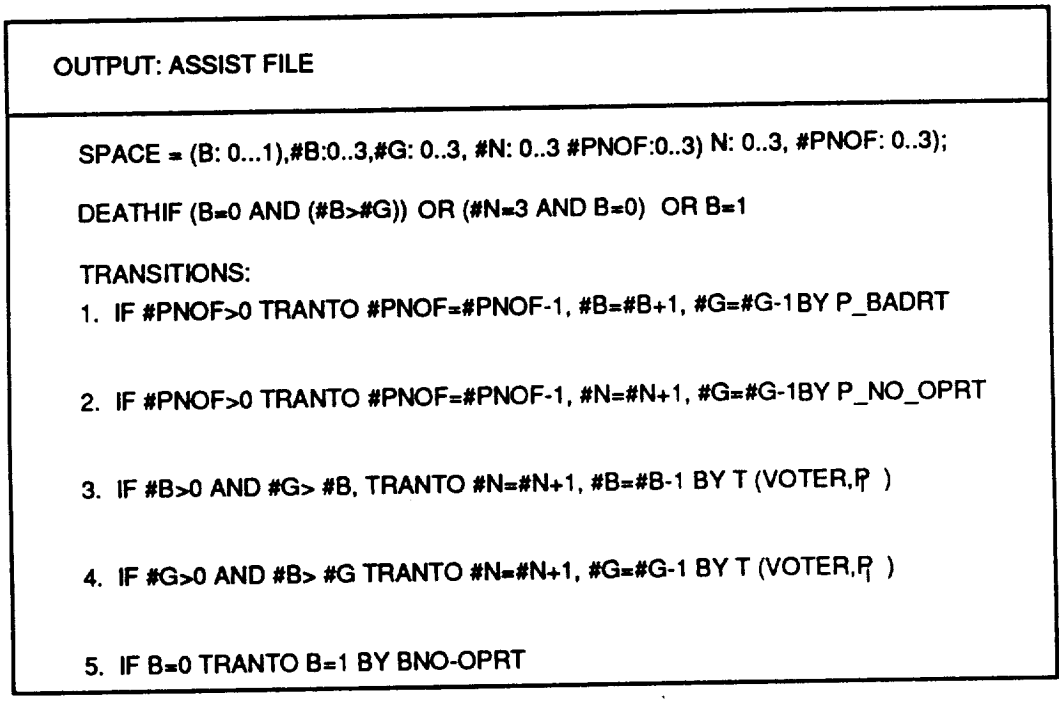


Figure D4. Example 2 Output

THIS IS A TRACE OF EXAMPLE 2 FOR ALL PROCESSES EXCEPT PROCESS3 AND PROCESS 2X WHICH ARE REFERENCED HERE WHEN INVOKED, BUT THE TRACES FOR THEIR ALGORITHMS ARE CONTAINED IN OTHER FILES. THE DOUBLE LINED ****** SEPARATE THE COMPONENT DOMAINS. THE ####### BOX ILLUSTRATES THE INFORMATION PASSED FROM THE PARENT COMPONENT TO ITS SUBCOMPONENTS.

EACH NUMBERED STEP INDICATES THE CONTROL FLOW OF THE PROCESSES FOR THIS EXAMPLE. THEY ARE NOT LISTED IN NUMERIC ORDER. RATHER, THE STEPS FOR EACH COMPONENT ARE LISTED TOGETHER TO ILLUSTRATE THE RECURSIVE TREE-LIKE BEHAVIOR OF THE ALGORITHMS. WHEN ONE COMPONENT CALLS ITS SUBCOMPONENT OR WHEN A SUBCOMPONENT PROCESS RETURNS TO ITS PARENT, THE STEP NUMBER IS GIVEN TO INDICATE WHAT TRACE STEP IS NEXT. THIS IS REPRESENTED BY A 'GOTO X' OR 'RETURN TO X' OR SIMPLY '(X)'.

FOR EACH STEP, THE PROCESS BEING INVOKED IS LISTED, AND EACH SUBSEQUENT NUMBER (OR LETTER) REFERENCES THE STEP IN ALGORITHM FOR THAT PROCESS.

```
**********************************************************************
**********************************************************************
1.  INITIALIZER:

    -highest level component is SYSTEM1
    -fill in table
    ##############################################################
    # TRACE TABLE:   SYSTEM1
    # function:              history        outside effects
    # input x;output y         none              none
    ##############################################################

    -pass control to SYSTEM1

**********************************************************************
**********************************************************************

2.  IN SYSTEM 1:  not lowest level
    PROCESS 1:
    1.  set up trace tables for components A and B
    ##############################################################
    # TRACE TABLE:  A
    # function               history       outside effects
    # input x; output x'       none             none
    ##############################################################
    # TRACE TABLE:  B
    # function               history       outside effects
    # input x'; output y       none             ----
    ##############################################################

    2.  set up priority Q (fifo)
    3.  call Process 2

    PROCESS 2:  take top element off list (A) and instantiate (3)

11.  IN SYSTEM1:
     PROCESS 2:
     1.  update external effects for component B
     2.  instantiate next subcomponent - B   pass to 12
         - returned output definitions and transitions for B
     3.  no more components - pass control to process 3 (16)

**********************************************************************
16.  PROCESS 3 in SYSTEM1:
TRANSITIONS:
GIVEN:
  FOR A:
       OUTPUT DEFINITION:   OUTPUT x'
                                 N IF #((x''i:(n))=3
                                 G IF #(x''i:(g)) > #(x''i:(b))
                                 B IF #(x''i:(b)) > #(x''i:(g))

      TRANSITIONS:
      1. IF PiNOF AND r=0 TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
```

```
                #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
    2. IF PiNOF AND r=0 TRANTO PiNO_OP, #(X''i:(N̄))=#(X''i:(N)+1),
                #(X''i:(G))=#(X''i:(Ḡ))-1  BY P_NO_OPRT
    3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''ī:(b̄))
       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(B))=#(X''i:(B))-1 BY T(voter,pi)
    4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(G))=#(X''i:(G))-1 BY T(voter,pi)

FOR B:
    TRANSITIONS:
                1.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

    OUTPUT DEFINITION:   for Y
                G IF x'(g) AND BNOF
                B IF x'(b) AND BNOF
                N IF (x'(n) AND BNOF) OR BNO-OP
 RESULT:
 ************************************************************
 * OUTPUT DEFINITION: for Y
 * OUTPUT B IF BNOF AND (#(x''i:(b)) > #(x''i(g)))
 * OUTPUT G IF BNOF AND (#(x''i:(g)) > #(x''i(b)))
 * OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP
 *
 * TRANSITIONS RETURNED:
 * 1. IF PiNOF AND r=0 TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
 *              #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
 * 2. IF PiNOF AND r=0 TRANTO PiNO_OP, #(X''i:(N̄))=#(X''i:(N)+1),
 *             #(X''i:(G))=#(X''i:(Ḡ))-1  BY P_NO_OPRT
 * 3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''ī:(b̄))
 *       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
 *             #(X''i:(B))=#(X''i:(B))-1 BY T(voter,pi)
 * 4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
 *       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
 *             #(X''i:(G))=#(X''i:(G))-1 BY T(voter,pi)
 * 5.  IF BNOF TRANTO BNO-OP BY BNO-OPRT
 ************************************************************
```

- see exam1.proc3.16.abs: process similarity is such that separate trace not documented

```
**********************************************************************
**********************************************************************
3.  IN COMPONENT A:  not lowest level
    PROCESS 1:
    1.  set up trace tables for components P and VOTER
    ##########################################################
    # TRACE TABLE:   P(i)
    # function              history       outside effects
    # input x; output x'';    none            none
    ##########################################################
    # TRACE TABLE:   VOTER
```

```
    # function                history          outside effects
    # input x'';output x'       none                ----
    ##########################################################

    2.  set up priority Q (fifo - rules 1 and 3)
    3.  call Process 2

    PROCESS 2:  take top element off list (P) and instantiate (4)
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    $ RETURNED FROM LOWER LEVEL: COMPONENT P
    $ TRANSITIONS:
    $   1.  IF PiNOF TRANTO PiBAD BY PBADRT
    $   2.  IF PiNOF TRANTO PiNO-OP BY PNO_OPRT
    $ OUTPUT DEFINITION:
    $   OUTPUT NIL IF PiNO_OP
    $   OUTPUT BAD IF PiBAD
    $   OUTPUT GOOD IF PiNOF
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

7.  IN COMPONENT A:
    PROCESS 2:
    1.  update other effects for VOTER -  rule 1
    ##########################################################
    # TRACE TABLE:   P(i)
    # function                history          outside effects
    # input x; output x'';     none               none
    ##########################################################
    # TRACE TABLE:   VOTER
    # function                history          outside effects
    # input x'';output x'       none               x''(i):(g,b,n)
    ##########################################################

    2.  Take top element off list (V) and instantiate (8)
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    $ RETURNED FROM LOWER LEVEL: COMPONENT VOTER
    $ TRANSITIONS: - none
    $ OUTPUT DEFINITION:
    $   OUTPUT x'
    $       n | all(x''i:(n)
    $       g | #(x''i:(g)) > #(x''i:(b))
    $       b | #(x''i:(b)) > #(x''i:(g))
    $   OUTPUT R
    $       0 | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
    $         | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
    $       1 | x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
    $         | x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

9.  IN COMPONENT A:
    PROCESS2:
    - finished with all subcomponents, pass to PROCESS3
**************************************************************
10. COMPONENT A: PROCESS3
    GIVEN P's
```

```
      for x': OUTPUT N IF PiNO_OP OR r=1
              OUTPUT B IF PiBAD AND r=0
              OUTPUT G IF PiNOF AND r=0
      IF PiNOF TRANTO PiBAD BY  PBADRT
      IF PiNOF TRANTO PiNO-OP BY PNO-OPRT

    VOTER'S   OUTPUT x'
                 n | all(x''i:(n)
                 g | #(x''i:(g)) > #(x''i:(b))
                 b | #(x''i:(b)) > #(x''i:(g))
              OUTPUT R
                 0 | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
                   | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
                 1 | x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
                   | x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))
   ***************************************************************
   *  RESULT:
   *  OUTPUT DEFINITION:   OUTPUT x'
   *                           N IF #((x''i:(n))=3
   *                           G IF #(x''i:(g)) > #(x''i:(b))
   *                           B IF #(x''i:(b)) > #(x''i:(g))
   *
   * TRANSITIONS:
   * 1. IF PiNOF AND r=0 TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
   *                #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
   * 2. IF PiNOF AND r=0 TRANTO PiNO_OP, #(X''i:(N))=#(X''i:(N)+1),
   *             #(X''i:(G))=#(X''i:(G))-1  BY P_NO_OPRT
   * 3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
   *       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
   *             #(X''i:(B))=#(X''i:(B))-1 BY T(voter,pi)
   * 4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
   *       TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
   *             #(X''i:(G))=#(X''i:(G))-1 BY T(voter,pi)
   ***************************************************************

- see exam2.proc3.10.abs for description of this step

*********************************************************************
*********************************************************************
4.  IN COMPONENT P:  lowest level
    PROCESS 1x:
    1.  create transitions for each fault possibility
        IF PiNOF TRANTO PiBAD BY PBADRT --> fault: BAD
        IF PiNOF TRANTO PiNO-OP BY PNO-OPRT --> fault: NO-OP
    2.  substitute into functional definition:
        fault BAD:
            OUTPUT x'':(b) from x | r=0
                   x'':(n)       | r=1   (* default *)
        fault NO-OP:
            OUTPUT x'':(n) from x | r=0
                   x'':(n)       | r=1
        nofault:   x'':(g) from x | r=0
                   x'':(n)       | r=1
    3.  Propagate:
```

```
            OUTPUT x'':(b)   | r=0 from fault BAD
                   x'':(n)   | r=1
            OUTPUT x'':(n)   | r=0 from fault NO-OP
                   x'':(n)   | r=1
            OUTPUT x'':(g)   | r=0
                   x'':(n)   | r=1
    4.  no history to analyze
    5.  output characteristics defined - don't call process 2x


5.
6.  PROCESS 3x:  Define output definition for each output characteristic
       OUTPUT (output characteristic)
                      IF (function define in 2x) and (fault state)

       OUTPUT N IF ((r=0 OR r=1) AND PiNO_OP) OR ((r=1) AND PiBAD)
                   OR (r=1 and PiNOF)

       - simplify: OUTPUT N IF  PiNO_OP OR r=1

       OUTPUT B IF (r=0) AND P_BAD
       OUTPUT G IF PiNOF AND r=0
    return to parent (A)   (7)


*************************************************************************
*************************************************************************
8.  IN COMPONENT VOTER:  lowest level
    PROCESS1X:
      - no faults to model (according to BBD) so only normal
        behavior to consider with external effects given

    PROCESS 2x:  -->  FA (P(i)):  input x''i FROM P(i)
                      output x' --> N | all x''i:(i)
                                      | FA z<>x', FA x''i:(^n):
                                            #(x''i=x') > #(x''i=z)
                      output r(i)  0 | x''i=x'
                                   1 | x''i<>x'
     FOR TRACE SEE PROCESS2X.VOTER

     BASICS:  take each  "|" as a separate domain
     1.  substitute outside data characteristic possibilities in function
     2.  apply rules to propagate characteristics through function
     3. for all resulting output characteristics
        - define output definition
        (*  at end, all non external data variables (such as z above)
        (*  will be gone, all quantifiers will be instantiated (gone)
        (*  all references to the output variable on right of | is gone

     RESULT:
         SEE TRACE TABLE

     return to A  (7)
```

```
**************************************************************************
**************************************************************************
12  IN B: lowest level
    PROCESS 1x:
      1.  create transitions for each fault possibility
          IF BNOF TRANTO PiNO-OP BY BNO-OPRT
      2.  substitute into functional definition:
          input x'; OUTPUT y:(n)
      3.  Propagate: no propagation necessary
      4.  no history to analyze
      5.  output characteristic defined: don't call process2x
                                  for NO_OP fault
      6.  call process2x for NOF state and x':(g,b,n) - goto 14
           - return from 14
     call process3x after all fault cases have been analyzed  - goto 15
13

14 PROCESS 2x in B for normal behavior
   GIVEN input x';output y from x'
         x':(g,b,n)

   UNSPECIFIED FUNCTION:
   y = O(x':(g,b,n))
   1.  for characteristic x':g
       - apply propagation rule: y=o(g) --> g
         OUTPUT y:(g) IF x':(g)
   2.  for characteristic x':b
       - apply propagation rule:      y = O(b) --> b
         OUTPUT y:(b) IF x':(b)
   3.  for characteristic x':n
       - apply propagation rule:  y = O(n) --> n
         OUTPUT y:(n) IF x':(n)

   no combining of common characteristics

15.  PROCESS 3x: Define output definition for each output characteristic
       OUTPUT (output characteristic)
                          IF (function define in 2x) and (fault state)

       OUTPUT Y:(N) IF () AND BNO-OP
                    OR x':(n) AND BNOF

       OUTPUT Y:(g) IF x':(g) AND BNO-OF

       OUTPUT Y:(b) IF x':(b) AND BNO-OF

    return to parent (system1 in step 11

**************************************************************************
PROCESS 4:
GIVEN:
  OUTPUT DEFINITION: for Y
     OUTPUT B IF #(x''i:(b)) > #(x''i(g))
     OUTPUT G IF #(x''i:(g)) > #(x''i(b))
```

```
    OUTPUT N IF (#(x''i:(n))=3) AND BNOF) OR BNO-OP

  TRANSITIONS:
     1. IF PiNOF AND r=0 TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
                  #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
     2. IF PiNOF AND r=0 TRANTO PiNO_OP, #(X''i:(N̄))=#(X''i:(N)+1),
                #(X''i:(G))=#(X''i:(Ḡ))-1  BY P_NO_OPRT
     3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''ī:(b̄))
        TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(B))=#(X''i:(B))-1 BY T(voter,pi)
     4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
        TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(G))=#(X''i:(G))-1 BY T(voter,pi)
     5.  IF BNOF TRANTO BNO-OP BY BNO-OPRT

1. APPLY MODEL REDUCTION TECHNIQUES:

    STEP 1:  find superfluous variables
        PiNO_OP,r=0

    STEP 2: combines superfluous variables for each component state
            and for each data variable
       - only 1 instance for each variable: can't combine

    STEP 3: heuristic d: for redundant components:
       - can't apply since PiBAD must be kept track of


  FINAL MODEL: no changes
 TRANSLATING INTO ASSIST:
 The reliability question was: R(y:(b) or Y:(n))
 Therefore the death state would be:

 DEATH IF:( BNOF AND (#(x''i:(b)) > #(x''i(g))) )
                     (* output definition for Y:(b) *)
           OR (#(x''i:(n))=3) AND BNOF) OR BNO-OP  )
                     (* output definition for Y:(n) *)

The variables will be represented as follows:
1.  a state vector element for each component state is created.  The
    number of values possible for the element will equal the number of
    states for the component with value 0 being the NOF state:
    For redundant components, the SYSD will determine the number
    of components: (* example:(SYSD say P(3) *) in exam1.

    B[0..1]     (* 0=BNOF, 1=BNO-OP *)
    P1[0..2]    (* 0=PNOF, 1=PBAD, and 2=PNO-OP *)
    P2[0..2]
    P3[0..2]

2.  For variables #(x''i:(b)), #(x''i(g))), and #(XI:(N)), a integer
    variable for each will be set up.

    #B: integer
```

#G: integer
#N: integer

3. other variables will be represented in the same manner as in 1. A state vector element for each variable will be created, and the number of values possible for the element will equal th number of values or characteristics for the variable:
R1[0..1]
R2[0..1]
R3[0..1]

The transitions and death state will be changed to reflect the representation:

```
DEATH IF:{ B=0  AND (#B > #G ) }
          OR (#N=3 AND B=0 )
          OR B=1


 TRANSITIONS:
    1a. IF P1=0 AND r=0 TRANTO P1=1, #B=#B+1, #G=#G-1  BY PBADRT
    1b. IF P2=0 AND r=0 TRANTO P2=1, #B=#B+1, #G=#G-1  BY PBADRT
    1c. IF P3=0 AND r=0 TRANTO P3=1, #B=#B+1, #G=#G-1  BY PBADRT

    2a. IF P1=0 AND r=0 TRANTO P1=2, #N=#N+1, #G=#G-1  BY PNO_OPRT
    2b. IF P2=0 AND r=0 TRANTO P2=2, #N=#N+1, #G=#G-1  BY PNO_OPRT
    2c. IF P3=0 AND r=0 TRANTO P3=2, #N=#N+1, #G=#G-1  BY PNO_OPRT

    3a. IF P1=1 and r=0 and #G > #B TRANTO R1=1, #N=#N+1, #B=#B-1
                            BY T(voter,pi)
    3b. IF P2=1 and r=0 and #G > #B TRANTO R2=1, #N=#N+1, #B=#B-1
                            BY T(voter,pi)
    3c. IF P3=1 and r=0 and #G > #B TRANTO R3=1, #N=#N+1, #B=#B-1
                            BY T(voter,pi)

    4a. IF P1=0 and r=0 and #B > #G TRANTO R1=1, #N=#N+1, #G=#G-1
                            BY T(voter,pi)
    4b. IF P2=0 and r=0 and #B > #G TRANTO R2=1, #N=#N+1, #G=#G-1
                            BY T(voter,pi)
    4c. IF P3=0 and r=0 and #B > #G TRANTO R3=1, #N=#N+1, #G=#G-1
                            BY T(voter,pi)

    5.  IF B=0 TRANTO B=1 BY BNO-OPRT
```

Note that this is a much bigger model that the one created in exam1. This is not so much because the problem was more complicated but rather because the model reduction techniques of process 4 that have thus been defined were not applicable to this model.

This is an illustration of the reasoning involved in process2x for the voter component in example 2.

```
FUNCTION:
        FA (P(n),@): input x''i:(n)FROM P(n);
        OUTPUT x' --> N | (ALL x''i:(n))
                        | FA z <> x', FA x''i:(n): #(x''i=x') > #(x''i=z)
         OUTPUT r(i)  0 | x''i=x'
                      1 | x''i<>x'
```

PROBLEM: Given the above function and the following input conditions:
x''i:(g,b,n)

Define the characteristics of the output as a function of its inputs

```
RESULT:
        OUTPUT x':(n)  | all(x''i:(n))
        OUTPUT x':(g)  | #(x''i:(g)) > #(x''i:(b))
        OUTPUT x':(b)  | #(x''i:(b)) > #(x''i:(g))
        OUTPUT R=0 | x''i:(n) and ALL(x''i:(n))
                   | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
                   | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
        OUTPUT R=1 | x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
                   | x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))
```

DISCUSSION: see exam1.proc2x.voter
STEPS: for the first two sentences:

```
        FA (P(n),@): input x''i:(n)FROM P(n);
        OUTPUT x' --> N | (ALL x''i:(n))
                        | FA z <> x', FA x''i:(n): #(x''i=x') > #(x''i=z)
```

the processing is the same as in exam1.proc2x.voter

```
SENTENCE 3:              OUTPUT r(i)  0 | x''i=x'
                                      1 | x''i<>x'
```

TAKE EACH DOMAIN SEPARATELY:
FIRST DOMAIN: OUTPUT r(i) 0 | x''i=x'

1. SUBSTITUTE IN KNOWN CHARACTERISTICS OF VARIABLES:
   A simple substitution similar to the previous substitutions will not suffice:

   OUTPUT r(i) 0 | x''i:(g,b,n)=y:(g,b,n)

   There is a dependency between the x' characteristics and the x''i characteristics which is missing in this definition. The output definition of x' shows this dependency:

```
x':(n)  | all(x''i:(n)
x':(g)  | #(x''i:(g)) > #(x''i:(b))
x':(b)  | #(x''i:(b)) > #(x''i:(g))
```

Therefore, in substituting in for x', the conditions underwhich x' has a characteristic must be explicitly noted:

1. substitute in for x':(n) and x''i:(g,b,n)

   OUTPUT r(i) 0 | x''i:(g,b,n)=x':(n) AND ALL(x''i:(n))

   The additional condition ALL(x''i:(n)) will reduce the possible characteristics of x''i from x''i:(g,b,n) to x''i:(n) and the resulting function will be:

   OUTPUT r(i) 0 | x''i:(n)=y:(n) and ALL(x''i:(n)

   (* note that the propagation of x' to x''i would also result
   (* x''i:(n)

2. substitute in for x':(g) and x''i:(g,b,n)

   OUTPUT r(i) 0 | x''i:(g,b,n)=y:(g) and #(x''i:(g)) > #(x''i:(b))

   PROPAGATION OF x' TO x''i:

   OUTPUT r(i) 0 | x''i:(g)=x':(g) and #(x''i:(g)) > #(x''i:(b))

3. substitute in for x':(b) and x''i:(g,b,n)

   OUTPUT r(i) 0 | x''i:(g,b,n)=y:(b) and #(x''i:(b)) > #(x''i:(g))

   PROPAGATION OF x' TO x''i:

   OUTPUT r(i) 0 | x''i:(b)=x':(b) and #(x''i:(b)) > #(x''i:(g))

The result of the substitutions, propagations:

```
OUTPUT r(i) 0 | x''i:(n)=y:(n) and ALL(x''i:(n)
OUTPUT r(i) 0 | x''i:(g)=y:(g) and #(x''i:(g)) > #(x''i:(b))
OUTPUT r(i) 0 | x''i:(b)=y:(b) and #(x''i:(b)) > #(x''i:(g))
```

Since x' is not the outputed variable in this sentence and it is not an input, it is an intermediate variable and therefore, it should be eliminated from the specification:

```
OUTPUT r(i) 0 | x''i:(n) and ALL(x''i:(n)
OUTPUT r(i) 0 | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
OUTPUT r(i) 0 | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
```

SECOND DOMAIN: OUTPUT r(i)=1 | x''i<>Y

1. SUBSTITUTE in for x':(n) and x''i:(g,b,n)

```
OUTPUT r(i) 1 | x''i:(g,b,n)<>x':(n) and ALL(x''i:(n))

PROPAGATE ALL(x''i:(n)) into x''i:(g,b,n)<>y:(n)
- OUTPUT r(i) 1 | x''i:(n)<>x':(n) and ALL(x''i:(n))
```

This specification is complete so far as the substitutions and propagations are concerned. However, it is irrelevant whether or not x''i <> x' if both (or either) variable is nil. Therefore this conditions can be considered false and the specification eliminated.

2. SUBSTITUTE in form x':(g) and x''i:(g,b,n):

```
OUTPUT r(i) 1 | x''i:(g,b,n)<>x':(g) and #(x''i:(g)) > #(x''i:(b))

PROPAGATE x' into x''i:
- OUTPUT r(i) 1 | x''i:(b,n)<>y:(g) and #(x''i:(g)) > #(x''i:(b))
```

3. SUBSTITUTE in form x':(b) and x''i:(g,b,n):

```
OUTPUT r(i) 1 | x''i:(g,b,n)<>y:(b) and #(x''i:(b)) > #(x''i:(g))

PROPAGATE x' into x''i:
- OUTPUT r(i) 1 | x''i:(g,n)<>y:(b) and #(x''i:(b)) > #(x''i:(g))
```

Again, the reference to x' may be eliminated so that the result for r(i)=0 and r(i)=1:

```
OUTPUT r(i) 0 | x''i:(n) and ALL(x''i:(n)
              | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
              | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
OUTPUT r(i) 1 | x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
              | x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))
```

The following is an overview of the processing involved in example2 when process3 is invoked. This example illustrates the reasoning behind the process flow (shown in small letters) and then shows a procedural algorithm that could be used to implement the reasoning (shown in ALL CAPITALS)

```
**************************************************************************
STEP 10 in exam1.trace.abs:  PROCESS 3 in component A:

COMPONENT A: PROCESS3
    GIVEN P's
        for x': OUTPUT N IF PiNO_OP OR r=1
                OUTPUT B IF PiBAD AND r=0
                OUTPUT G IF PiNOF AND r=0
        IF PiNOF TRANTO PiBAD BY  PBADRT
        IF PiNOF TRANTO PiNO-OP BY PNO-OPRT

     VOTER'S   OUTPUT x'
                   n | all(x''i:(n)
                   g | #(x''i:(g)) > #(x''i:(b))
                   b | #(x''i:(b)) > #(x''i:(g))
               OUTPUT R
                   0 | x''i:(g) and #(x''i:(g)) > #(x''i:(b))
                     | x''i:(b) and #(x''i:(b)) > #(x''i:(g))
                   1 | x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
                     | x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))

DESIRED OUTPUT:

    OUTPUT DEFINITION:   OUTPUT x'
                             N IF #((x''i:(n))=3
                             G IF #(x''i:(g)) > #(x''i:(b))
                             B IF #(x''i:(b)) > #(x''i:(g))

     TRANSITIONS:
     1. IF PiNOF AND r=0 TRANTO PiBAD, #(X''i:(B))=#(X''i:(B)+1),
                #(X''i:(G)=#(X''i:(G))-1  BY P_BADRT
     2. IF PiNOF AND r=0 TRANTO PiNO_OP, #(X''i:(N))=#(X''i:(N)+1),
                #(X''i:(G))=#(X''i:(G))-1  BY P_NO_OPRT
     3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
        TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(B))=#(X''i:(B))-1 BY T(voter,pi)
     4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
        TRANTO r=1, #(X''i:(N))=#(X''i:(N))+1,
                #(X''i:(G))=#(X''i:(G))-1 BY T(voter,pi)

**************************************************************************
DISCUSSION:  SEE EXAM1.PROC3.10.ABS
**************************************************************************

STEP 1:  Look at the subcomponent that produces the final
output:  the voter.
```

VOTER'S DEFINITION:
OUTPUT B IF #(x''i:(B)) > #(x''i:(G))

ANALYZE EACH POSSIBLE OUTPUT CHARACTERISTIC SEPARATELY:
1. OUTPUT B IF #(x''i:(B)) > #(x''i:(G)):

ANALYZE THE CONDITION (#(x''i:(B)) > #(x''i:(G))) UNDER WHICH THE OUTPUT CHARACTERISTIC (B) CAN OCCUR
TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- #(x''i:(B)) > #(x''i:(G)): find the transitions that affect this condition

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY

1. #(x''i:(B)) : find transitions that affect this condition:

The transitions that contribute to the number of x inputs being bad is directly related to the transitions that contribute to one x value being bad; find transitions that contribute to x being bad:

FOR FUNCTIONS SUCH AS #, THERE HAS TO BE 'KNOWLEDGE' IN THE SYSTEM TO DETERMINE WHAT CONDITION TO ANALYZE (CALLED CONDITION OF INTEREST - COI) ++ 3 ++

x''i:(b): find transitions that affect this conditions:

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT B IF PiBAD AND r=0

ANALYZE THE CONDITION (PiBAD AND r=0) UNDER WHICH THE OUTPUT CHARACTERISTIC (B) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY

1. PiBAD - find the transitions that affect this condition
P fails by fault BAD (as specified in the BBD definition of P). Therefore the transition that contributes to PiBAD is:

P's TRANSITIONS:
1. IF PiNOF TRANTO PiBAD BY PiBADRT

WHEN A INPUT CHARACTERISTIC (x':(b)) HAS BEEN ANALYZED AND TRANSITIONS HAVE BEEN FOUND,

EITHER THE TRANSITIONS WILL HAVE TO BE CHANGED TO REFLECT AN AFFECT ON THE INPUT CHARACTERISTIC OR THE OUTPUT DEFINITION BEING ANALYZED WILL HAVE TO BE MODIFIED TO REFERENCE THAT SAME VARIABLES AS THE TRANSITIONS REFERENCE. ++ 4 ++

This transition contributes to PiBAD, but the COI is x''i:(b). Therefore, we must change the transition to reflect a change in x''i. Since P's output definition states that the output is b if PiBAD and R=0, the following change to the transition is made:

OUTPUT x''i:(b) IF PiBAD AND r=0
IF PiNOF TRANTO PiBAD BY PiBADRT.

IF ... TRANTO ..., x''i:(b)

1. IF PiNOF AND r=0 TRANTO PiBAD, x''i:(b)
   BY PiBADRT

2. r=0 - find the transitions that affect this condition

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

VOTER'S OUTPUT DEFINITION:
OUTPUT R=0 IF x''i:(g) and
#(x''i:(g)) > #(x''i:(b)) OR
x''i:(b) and #(x''i:(b)) > #(x''i:(g))

this output definition could be analyzed in the same manner as presented above. However, note that r is defined in terms of the variable characteristic x''i:(b) which is a condition currently under analysis (r=0 is a condition that is being analyzed in order to find the transitions that contribute to x''i:(b). This indicates that x''i:(b) is dependent on the condition r=0 which is dependent on the condition x''i:(b), and therefore in the domain:

OR x''i:(b) and #(x''i:(b)) > #(x''i:(g))

the condition x''i:(b) need not be analyzed further since there is no transition to x''i:(b) indicated.

The condition #(x''i:(b)) is currently being

analyzed and therefore is ignored in further analysis also.

IF A VARIABLE CHARACTERISTIC IN THE CONDITION PART OF THE OUTPUT DEFINITION IS A CHARACTERISTIC CURRENTLY BEING ANALYZED, THAT CONDITION NEED NOT BE ANALYZED FURTHER: ++ 1 ++ rule 1

Therefore, the remaining conditions to be analyzed for r=0:

x''i:(g) and #(x''i:(g))

ANALYZE THE SUBCONDITIONS SEPARATELY

1. x''i:(g):

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT x''i:(g) IF PiNOF AND r=0

IF A VARIABLE CHARACTERISTIC IN THE CONDITION PART OF THE OUTPUT DEFINITION IS A CHARACTERISTIC CURRENTLY BEING ANALYZED, THAT CONDITION NEED NOT BE ANALYZED FURTHER: ++ 1 ++ rule 1

r=0 currently being defined, discard condition:

PiNOF - no transitions contribute

2. #(x''i:(g)):

COI: x''i:(g) - already analyzed
no transitions

RESULT OF ANALYZING VOTERS OUTPUT DEFINITION FOR r=0: no transitions found

- thus, in analyzing OUTPUT x''i:(B) IF PiBAD AND r=0 only one transition has been found

1. IF PiNOF AND r=0 TRANTO PiBAD, x''i:(b)
BY PBADRT

Recall that x''i:(b) was analyzed as a COI for the analysis of #(x''i:(b)). Any transitions found to affect x''i:(b) must be changed to reflect a change to #(x''i:(b)).

WHEN A INPUT CHARACTERISTIC (x':(b)) HAS BEEN ANALYZED AND TRANSITIONS HAVE BEEN FOUND, EITHER THE TRANSITIONS WILL HAVE TO BE CHANGED TO REFLECT AN AFFECT ON THE INPUT CHARACTERISTIC OR THE OUTPUT DEFINITION BEING ANALYZED WILL HAVE TO BE MODIFIED TO REFERENCE THAT SAME VARIABLES AS THE TRANSITIONS REFERENCE. ++ 4 ++

- change x''i:(b) to #(x''i:(b)=#(x''i:(b))+1

1. IF PiNOF AND r=0 TRANTO PiBAD,
   #(x''i:(B)=#(x''i:(b))+1 BY PBADRT

2. #(x''i:(g)) :

Even though this condition has been analyzed within the the analysis of #(x''i:(b)), that analysis was curtailed because of condition r=0 being re-encountered. However, in analysis of #(x''i:(g)), there is no prior condition r=0, and therefore, #(x''i:(g)) must be reanalyzed.

WHEN CONDITION HAS ALREADY BEEN ANALYZED, A CHECK IS MADE TO DETERMINE IF ANY RESTRICTIONS ON ANALYSIS WERE IMPOSED TO SHORTEN THE ANALYSIS. IF SUCH RESTRICTIONS ARE NOT APPLICABLE IN THE NEW DOMAIN UNDER WHICH THE CONDITION IS TO BE ANALYZED, THE CONDITION MUST BE REANALYZED WITHOUT SUCH RESTRICTIONS. ++ 2 ++

FOR FUNCTIONS SUCH AS #, THERE HAS TO BE SOME 'KNOWLEDGE' IN THE SYSTEM TO DETERMINE WHAT CONDITION TO ANALYZE

x''i:(g): find transitions that affect this conditions:

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME OUTPUT CHARACTERISTIC IN THE COMPONENT THAT OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT G IF PiNOF AND r=0

ANALYZE THE CONDITION (PiNOF AND r=0) UNDER WHICH THE OUTPUT CHARACTERISTIC (G) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- PiNOF is a no fault state and there is no transition that leads to this state.

- r=0 this condition has already been analyzed, and no transitions have been found

Since no transitions that contribute to x''i:(g),

there are no transitions that contribute to #(x''i:(g)). However, only transitions that contribute to #(x''i:(g)) have been analyzed. To determine transitions that detract from #(x''i:(g)), look at transitions that have already been defined as contributing to other conditions and determine if these transitions detract from #(x''i:(g)). Transition #1 does detract from #(x''i:(g)) and so it must be changed to reflect this:

1. IF PiNOF AND R=0 TRANTO PiBAD,#(X''i:(B)=#(X''i:(B))+1, #(X''i:(G)=#(X''i:(G))-1 BY P_BADRT

ALL PREVIOUSLY DEFINED TRANSITIONS THAT DETRACT FROM THE CURRENT CONDITION ARE CHANGED TO REFLECT THIS? THERE MUST BE A WAY FOR THE PROCESS 3 TO DETECT WHEN A PREVIOUSLY DEFINED TRANSITION DETRACTS FROM A CURRENT CONDITITION ++ 5 ++

Therefore in analyzing the subconditions #(x''i:(b)) and #(x''i:(g)), we have found 1 transition to contribute to #(x''i:(b)) > #(x''i:(g)):

1. IF PiNOF AND R=0 TRANTO PiBAD,#(x''i:(B)=#(x''i:(b))+1 #(X''i:(G)=#(X''i:(G))-1 BY PBADRT

2. OUTPUT N IF ALL(X''i:(n)):

ANALYZE THE CONDITION UNDER WHICH THE OUTPUT CHARACTERISTIC CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

- ALL(X''i:(n)): find the transitions that affect this condition

One way in which analysts represent the transition to a system state in which all x inputs are n is to keep track of the number of x inputs that are n and to determine when that number equals the total number of possible x values. Therefore, an analyst would change the condition ALL(X''i:(n)) to #(x''i:(n)) = 3) where 3 is the total number of x inputs as determined by the BBD specification. Therefore, the analyst tries to find the transitions that contributes to #(x''i:(n)):

FOR SOME OPERATIONS (SUCH AS THE OPERATION 'ALL') THE REPRESENTATION OF THE CONDITION IS CHANGED IN ORDER TO MODEL IT LATER. ++ 3 ++

- #(x''i:(n)): find the transitions that affect this condition

As in #(x''i:(b)) and #(x''i:(g)), find the transitions that contribute to x''i:(n) and then change any transitions to reflect a contribution to #(x''i:(n))

- x''i:(n): find the transitions that affect this condition:

FOR INPUT CHARACTERISTICS, ANALYZE THE SAME
OUTPUT CHARACTERISTIC IN THE COMPONENT THAT
OUTPUTED THE DATA

P's OUTPUT DEFINITION:
OUTPUT N IF PiNO_OP OR R=1

ANALYZE THE CONDITION (PiNO_OP) UNDER WHICH THE OUTPUT CHARACTERISTIC (N) CAN OCCUR TO DETERMINE THE TRANSITIONS THAT AFFECT THE CONDITION:

1. PiNO_OP is a fault state of component P that results when P fails by fault NO_OP (as specified in the BBD definition of P). Therefore, the transition that contributes to PiNO_OP is:

P's TRANSITION:
2. IF PiNOF TRANTO PiNO_OP BY PiNO_OPRT

THE TRANSITION IS CHANGED TO REFLECT THE CHANGE TO THE CONDITION BEING ANALYZED (x''i:(n)) ++ 4 ++
This output definition is OUTPUT x''i:(n) IF PiNO-OP OR r=1. To change the transition to reference a change TO x''i:(n), the clause AND NOT(r=1) must be added since according to the output definition if R=1 then x''i:(n).

OUTPUT x''i:(n) IF PiNO_OP OR R=1
2. IF PiNOF TRANTO PiNO_OP BY PiNO_OPRT

2. IF ... TRANTO ..., x''i:(n)

2. IF PiNOF AND NOT(r=1) TRANTO PiNO_OP, x''i:(n)
BY PiNO_OPRT
2. IF PiNOF AND r=0 TRANTO PiNO_OP, x''i:(n) BY PiNO_OPRT

2. R=1 - find the transitions that affect this condition:
FOR INPUT CHARACTERISTICS, ANALYZE THE SAME
OUTPUT CHARACTERISTIC IN THE COMPONENT THAT
OUTPUTED THE DATA

VOTER'S OUTPUT DEFINITION:
OUTPUT r=1 IF x''i:(b,n) and #(x''i:(g)) > #(x''i:(b))
OR IF x''i:(g,n) and #(x''i:(b)) > #(x''i:(g))

This can be changed to what will be called OR domain form in which each domain is delineated by the logical OR:

```
OUTPUT r=1 IF x''i:(b) and #(x''i:(g)) > #(x''i:(b))
           OR x''i:(n) and #(x''i:(g)) > #(x''i:(b))
           OR x''i:(g) and #(x''i:(b)) > #(x''i:(g))
           OR x''i:(n) and #(x''i:(b)) > #(x''i:(g))
```

NOTE: the condition x''i:(n) is a condition current under analysis (r=1 is a condition that is being analyzed in order to find the transitions that contribute to x''i:(n)). This indicates that x''i:(n) is dependent on the condition r=1 which is dependent on the condition x''i:(n), and therefore the domain:

```
   OR x''i:(n) and #(x''i:(g)) > #(x''i:(b))
```

and the domain:

```
   OR x''i:(n) and #(x''i:(b)) > #(x''i:(g))
```

need not be analyzed further since there is no transition to x''i:(n) indicated.

IF A VARIABLE CHARACTERISTIC IN THE CONDITION PART OF THE OUTPUT DEFINITION IS A CHARACTERISTIC CURRENTLY BEING ANALYZED, THE CONDITION IS NOT ANALYZED FURTHER.
++ 1 ++ rule 2

The second and third domains have no new conditions to analyze. The remaining domains to analyze are:

```
OUTPUT r=1 IF x''i:(b) and #(x''i:(g)) > #(x''i:(b))
           OR x''i:(g) and #(x''i:(b)) > #(x''i:(g))
```

The conditions x''i:(b) and x''i:(g) have previously been analyzed. Substituting the logical equivalent of these conditions ((PiBAD and r=0), (PiNOF and r=0) respectively):

```
OUTPUT r=1 IF (PiBAD and r=0) and #(x''i:(g))>#(x''i:(b))
           OR (PiNOF and r=0) and #(x''i:(b))>#(x''i:(g))

 EQUIVALENT:
   OUTPUT r=1 IF PiBAD and r=0 and #(x''i:(g))>#(x''i:(b))
              OR PiNOF and r=0 and #(x''i:(b))>#(x''i:(g))
```

Note that each domain contains an IF ... AND r=0 clause when the output is OUTPUT r=1. This indicates a transition from r=0 to r=1, but the transition is not fault related. Rather, it is a system reaction to a fault, or a recovery to a fault. This output definition states that if r=0 and if component P fails by fault BAD and if the majority of the inputs to the voter are good then the output of the voter r will be 1. To model this as a transition,

3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
TRANTO r=1 by T(voter,Pi)

where T(voter,Pi) is the time for all components in the cycle to execute their functions. This would be a fast transition relative to the rates for faults, but the rates for component faults and the fast transition rates are defined by the analyst in the BBD and used after the abstract model is defined and inputed into a Reliability Analysis Tool (SURE).

The second domain would also be converted into a non-fault transition:

4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
TRANTO r=1 by T(voter,pi)

IF THE CONDITION PART OF AN OUTPUT DEFINITION INCLUDES A NOT(effect) FOR THE EFFECT BEING DEFINED, A NON-FAULT TRANSITION IS CREATED ++ 1 ++ rule 3

THE TRANSITIONS ARE CHANGED TO REFLECT THE CHANGE TO THE CONDITION BEING ANALYZED x''i:(n)

OUTPUT x''i:(n) IF PiNO_OP OR R=1
3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
TRANTO r=1 by T(voter,pi)
4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
TRANTO r=1 by T(voter,pi)

IF ... TRANTO ... AND x''i:(n)

3. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
AND NOT(r=1) TRANTO r=1, x''i:(n)
by T(voter,pi)

4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
TRANTO r=1, x''i:(n) by T(voter,pi)

TRANSITION IS CHANGED TO REFLECT THE CHANGE TO CONDITION BEING ANALYZED #(x''i:(n)) ++ 4 ++

- CHANGE x''i:(n) TO #(x''i:(n)) = #(x''i:(n)) + 1

2. IF PiNOF AND r=0 TRANTO PiNO-OP,
#(x''i:(n)=#(x''i:(n))+1
BY P_NO_OPRT
3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
TRANTO r=1, #(x''i:(n)=#(x''i:(n))+1 by T(voter,pi)
4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
TRANTO r=1, #(x''i:(n)=#(x''i:(n))+1 by T(voter,pi)

- Again, look at previously defined transitions and determine whether or not they detract from #(x''i:(n)): transition #1 does not. But notice that the newly defined transition #2 detracts from #(x''i:(g)). Therefore this transition must be changed to reflect the affect on #(x''i:(g)):

ANALYZE ALL PREVIOUSLY DEFINED TRANSITIONS FOR A DETRACTORY EFFECT ON CONDITION CURRENTLY BEING ANALYZED: ++ 5 ++

- none

ANALYZE ALL CURRENTLY DEFINED TRANSITIONS FOR A DETRACTORY EFFECT ON PREVIOUSLY DEFINED CONDITIONS ++ 5 ++

2. IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1, #(X''i:(g)=#(X''i:(g))-1, BY P_NO_OPRT

3. OUTPUT g IF #(x''i:(g)) > #(x''i:(b)) :

DIVIDE EACH CONDITION INTO ITS SUBCONDITIONS AND ANALYZE THE SUBCONDITIONS SEPARATELY:

- #(x''i:(g)): find the transitions that affect this condition:
  - this condition has already been analyzed

- #(x''i:(b)): find the transitions that affect this condition:
  - this condition has already been analyzed

  - no more transitions found

The final output for this process is:

OUTPUT DEFINITION:
OUTPUT B IF #(x''i:(b)) > #(x''i:(g))
OUTPUT N IF #(x''i:(n))=3)
OUTPUT G IF #(x''i:(g)) > #(x''i:(b))

TRANSITIONS:

1. IF PiNOF TRANTO PiBAD, #(x''i:(b)=#(x''i:(b))+1, #(x''i:(g)=#(x''i:(g))-1 BY P_BADRT

2. IF PiNOF TRANTO PiNO-OP, #(X''i:(N)=#(X''i:(N))+1, #(X''i:(g)=#(X''i:(g))-1, BY P_NO_OPRT

```
   3. IF PiBAD and r=0 and #(x''i:(g)) > #(x''i:(b))
         TRANTO r=1, #(x''i:(n)=#(x''i:(n))+1 by T(voter,pi)

   4. IF PiNOF and r=0 and #(x''i:(b)) > #(x''i:(g))
         TRANTO r=1, #(x''i:(n)=#(x''i:(n))+1 by T(voter,pi)

         #(x''i:(g)=#(x''i:(g))-1  BY P_BADRT

2. IF PiNOF TRANTO PiBAD,#(x''i:(N)=#(x''i:(N))+1,
          #(x''i:(g)=#(x''i:(g))-1
                          BY P_BADRT
```

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182005		

4. Title and Subtitle

RELIABILITY MODEL GENERATOR SPECIFICATION

5. Report Date

MARCH 1990

6. Performing Organization Code

7. Author(s)

Gerald C. Cohen
Catherine M. McCann

8. Performing Organization Report No.

10. Work Unit No.

505-66-71-02

9. Performing Organization Name and Address

Boeing Advanced Systems
P.O. Box 3707, M/S 33-12
Seattle, WA 98124-2207

11. Contract or Grant No.

NAS1-18099

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

NASA Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Daniel L. Palumbo
Final Report

16. Abstract

This report describes the Reliability Model Generator, a program which produces reliability models from block diagrams for ASSIST, the interface for the reliability evaluation tool SURE. The report gives an account of motivation for RMG and discusses the implemented algorithms. The appendices contain the algorithms and two detailed traces of examples.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
SURE, ASSIST, reliability models, local reliability models, output characteristic, FMEA, failure mode	Unclassified - Unlimited Subject Category 66

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
UNCLASSIFIED	UNCLASSIFIED	247	A11

NASA FORM 1626 OCT 86